MACHINE LEARNING WITH PYTHON

By SAI LIKHITH PANUGANTI
https://www.linkedin.com/in/sailikhith

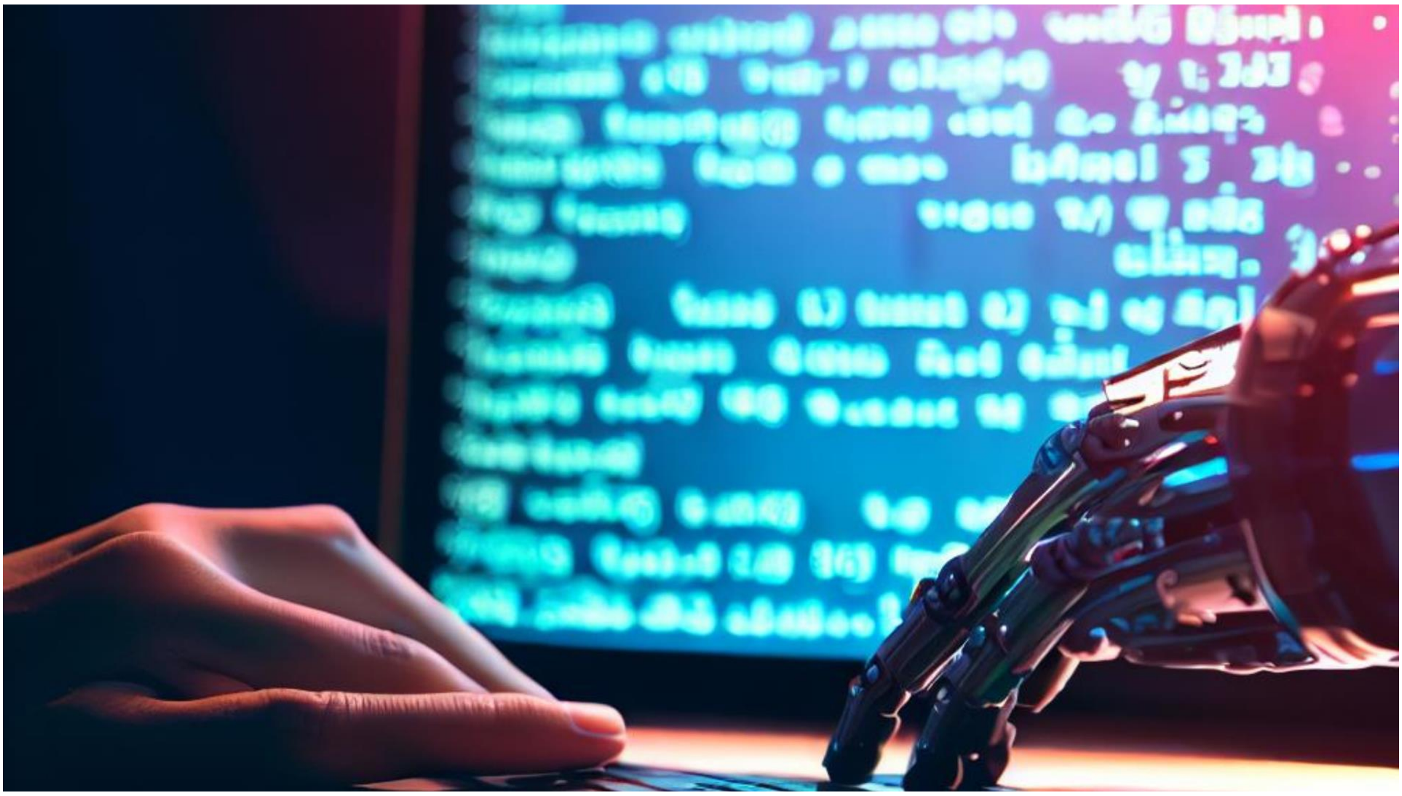


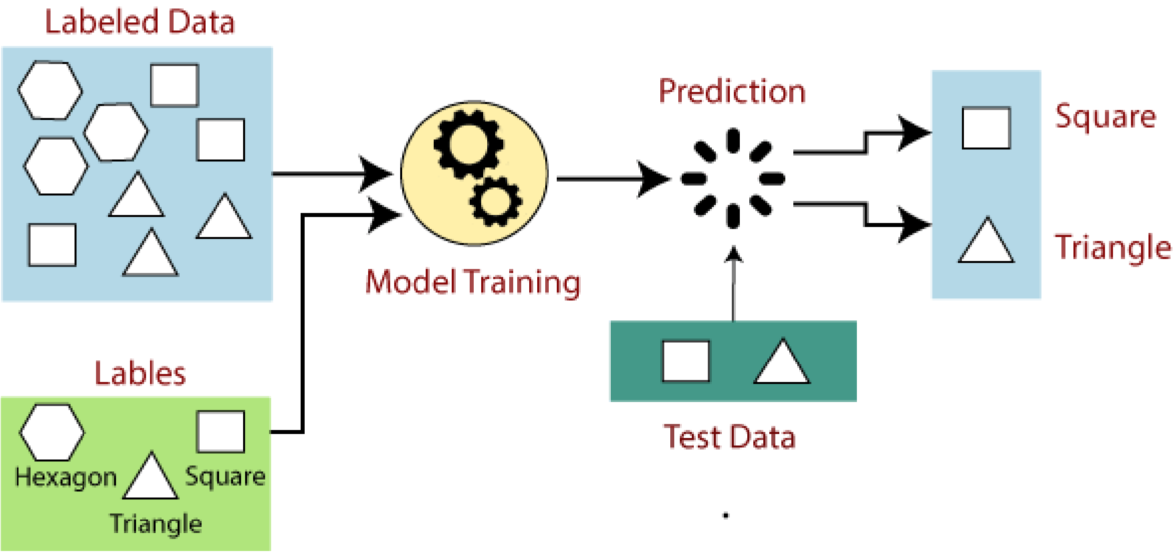
Table of Contents

Introduction to Machine Learning

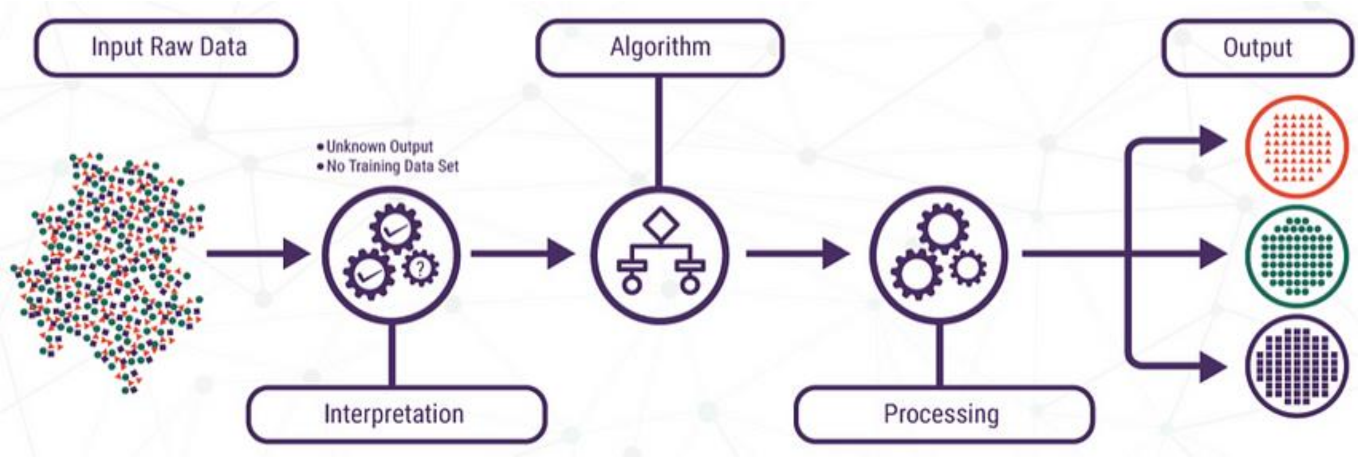
Machine Learning is a field of study and application that involves developing algorithms and models that allow computers to learn and make predictions or decisions without being explicitly programmed. It is a subset of Artificial Intelligence (AI) and is based on the idea that machines can learn from data and improve their performance over time.

Theory of Machine Learning:

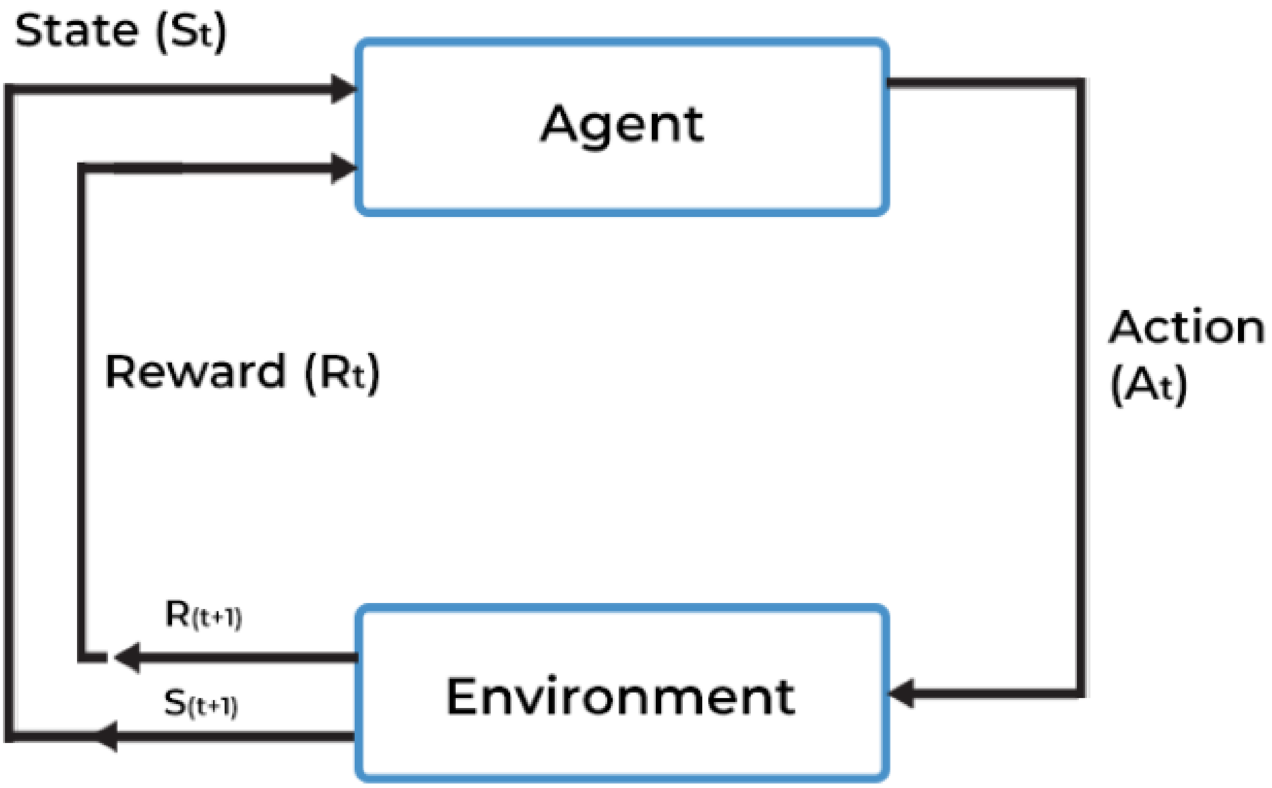
1. **Supervised Learning:** In this approach, the machine learns from labelled data, where each input example is associated with a corresponding target value. The goal is to learn a mapping function that can predict the target value for new, unseen inputs.



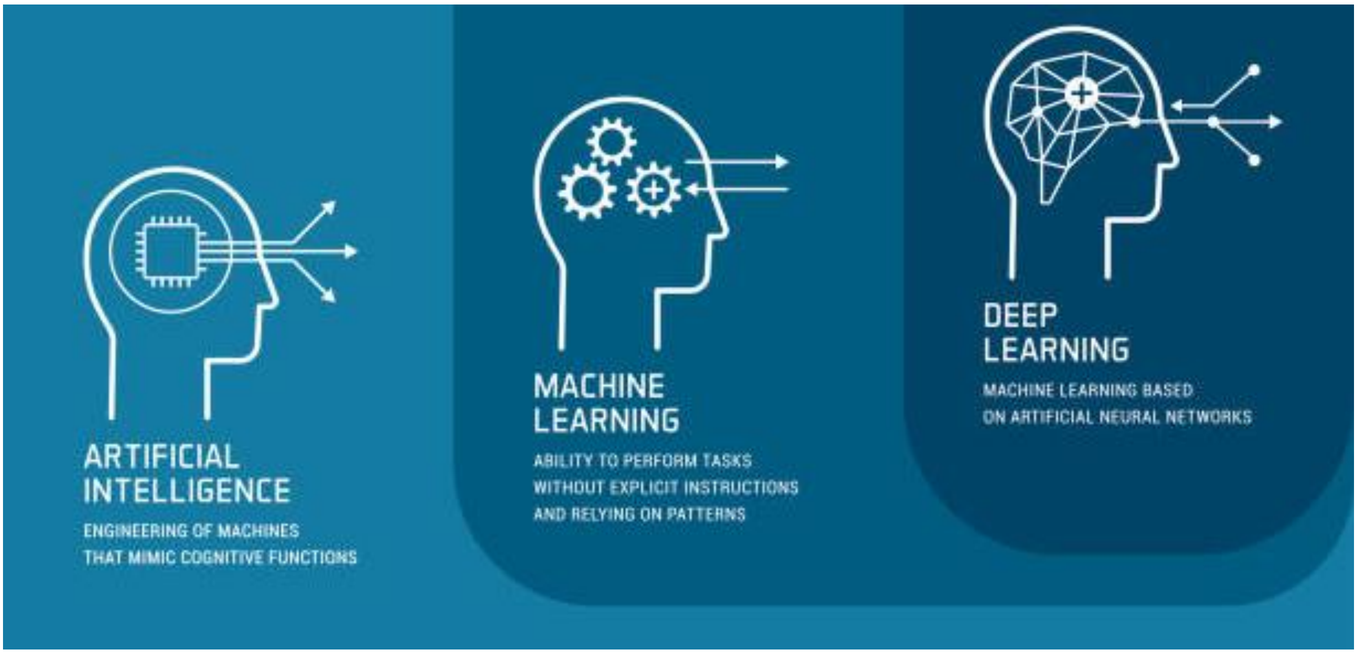
2. **Unsupervised Learning:** In unsupervised learning, the machine learns from unlabelled data, seeking to discover patterns or relationships within the data without any specific target variable. Clustering and dimensionality reduction are common unsupervised learning techniques.



3. **Reinforcement Learning:** This approach involves training an agent to interact with an environment and learn optimal actions through trial and error. The agent receives feedback in the form of rewards or penalties based on its actions.



4. **Deep Learning:** Deep Learning is a subfield of Machine Learning that focuses on the development of neural networks with multiple layers. These networks are capable of learning complex patterns and representations from data.


ARTIFICIAL INTELLIGENCE
ENGINEERING OF MACHINES THAT MIMIC COGNITIVE FUNCTIONS
MACHINE LEARNING
ABILITY TO PERFORM TASKS WITHOUT EXPLICIT INSTRUCTIONS AND RELYING ON PATTERNS
DEEP LEARNING
MACHINE LEARNING BASED ON ARTIFICIAL NEURAL NETWORKS

Advantages of machine learning:

- **Automation:** Machine Learning enables automation of tasks that would otherwise require manual effort and decision-making.
- **Handling Complex Data:** ML algorithms can handle and extract insights from large and complex datasets.
- **Improved Accuracy:** ML models can make accurate predictions or decisions based on patterns and relationships found in the data.
- **Adaptability:** Machine Learning models can adapt and improve their performance over time as new data becomes available.

Disadvantages of machine learning:

- **Data Dependency:** Machine Learning models heavily rely on the quality and quantity of available data. Insufficient or biased data can lead to inaccurate or unfair predictions.
- **Lack of Explain ability:** Some ML algorithms, such as deep neural networks, are often considered black boxes, making it difficult to understand the reasoning behind their decisions.
- **Overfitting:** ML models may become overly specialized in the training data, leading to poor performance on new, unseen data.
- **Computational Requirements:** Complex ML models, especially deep learning models, require significant computational resources and training time.

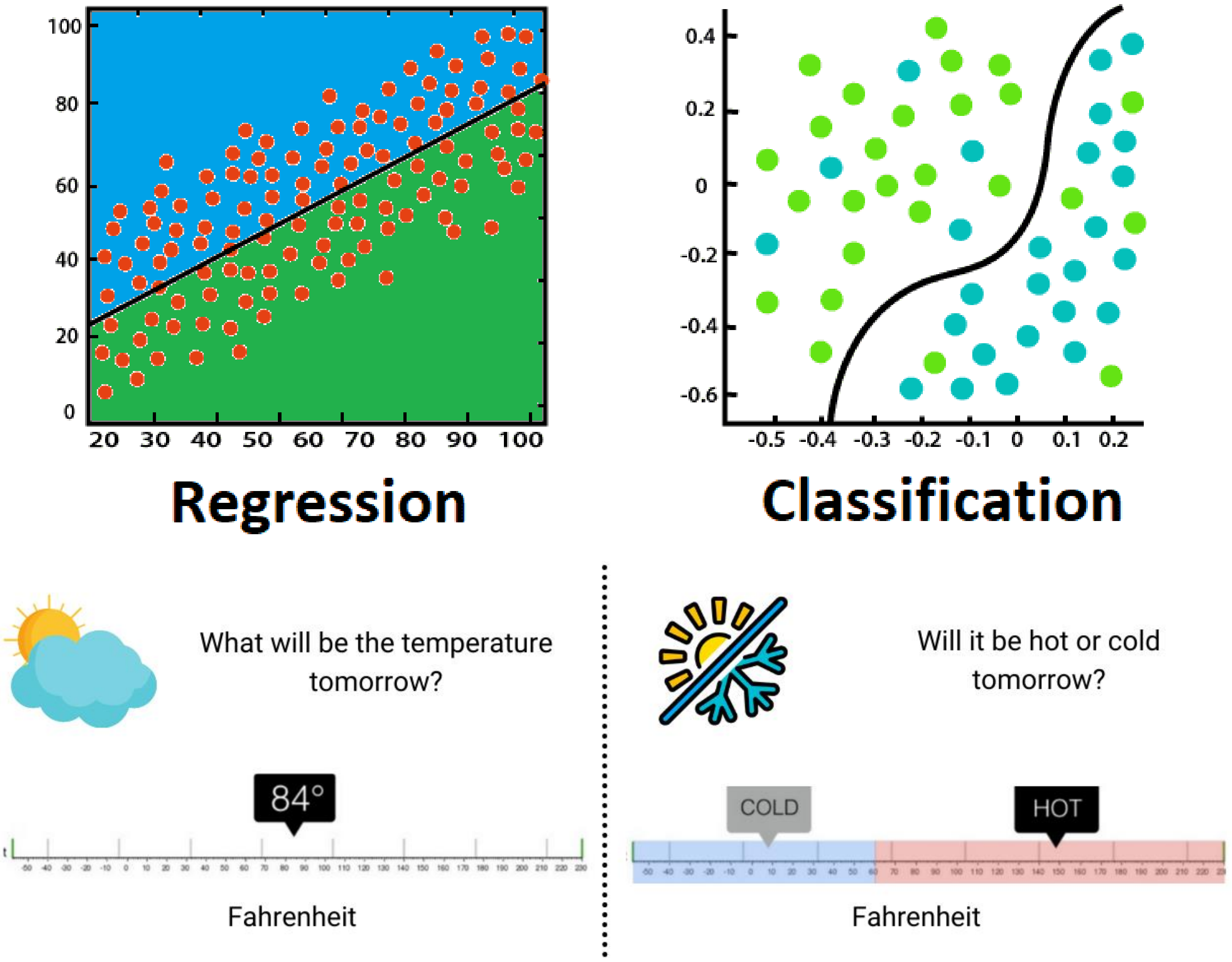
Applications of machine learning:

- **Image and Speech Recognition:** ML algorithms are used in applications like facial recognition, object detection, and speech-to-text conversion.
- **Natural Language Processing:** ML techniques power language translation, sentiment analysis, chatbots, and text generation.
- **Recommendation Systems:** ML-based recommendation systems are used in e-commerce, streaming services, and content personalization.
- **Fraud Detection:** ML models can detect patterns of fraudulent behavior in financial transactions and flag suspicious activities.

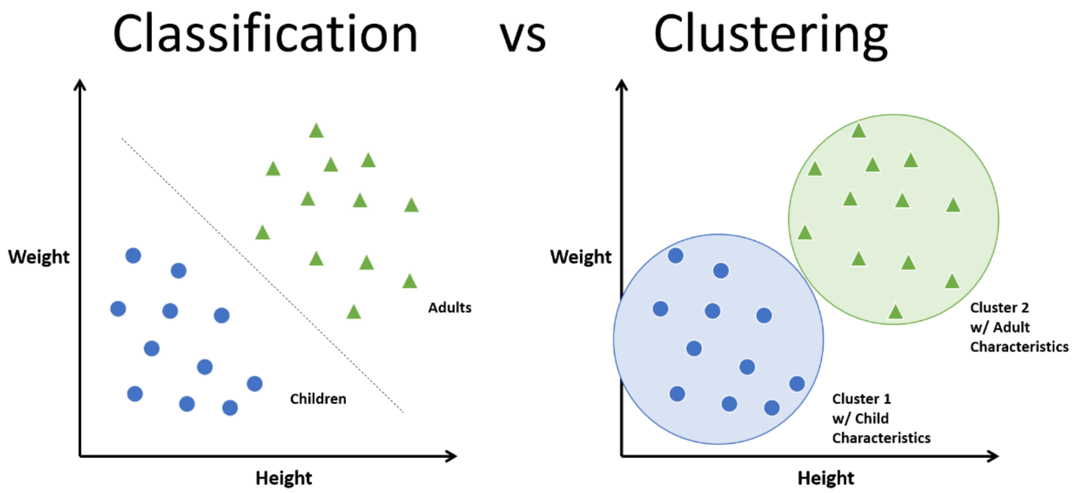
- **Healthcare:** Machine Learning is applied in disease diagnosis, drug discovery, personalized medicine, and medical imaging analysis.
- **Autonomous Vehicles:** ML algorithms are used in self-driving cars to interpret sensor data, make driving decisions, and improve safety.

Key topics in machine learning:

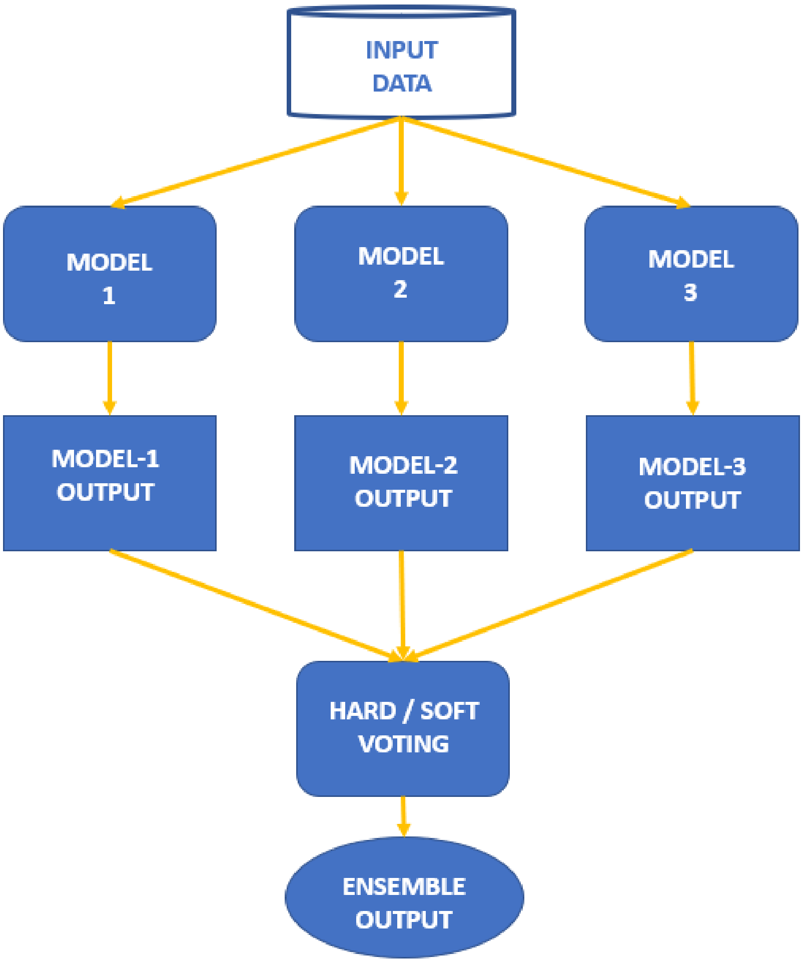
- **Evaluation Metrics:** Evaluation metrics measure the performance of ML models. Common metrics include accuracy, precision, recall, F1 score, mean squared error (MSE), and area under the receiver operating characteristic curve (AUC-ROC).
- **Regression:** Regression algorithms are used to predict continuous numerical values based on input variables.
- **Classification:** Classification algorithms are used to assign input examples to predefined categories or classes.



- **Clustering:** Clustering algorithms group similar data points together based on their characteristics.


Classification vs Clustering
Weight
Height
Adults
Children
Weight
Height
Cluster 2 w/ Adult Characteristics
Cluster 1 w/ Child Characteristics

- **Dimensionality Reduction:** These techniques aim to reduce the number of input variables while preserving important information.
- **Ensemble Learning:** Ensemble learning combines multiple ML models to make predictions or decisions. It improves accuracy, reduces overfitting, and includes techniques like bagging, boosting, and stacking.



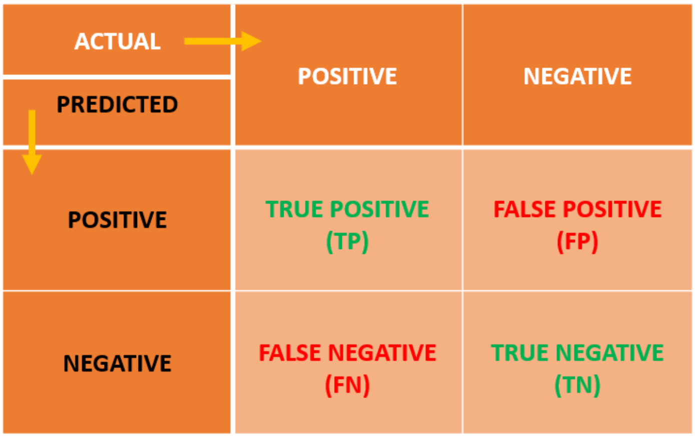
Evaluation Metrics

Confusion Matrix:

A confusion matrix summarizes the performance of a classification model by tabulating the counts of true positives, false positives, true negatives, and false negatives. It is useful for understanding the types of errors made by the model and assessing its performance across different classes.

The confusion matrix is typically organized with the following side headings:

- **True Positives (TP):** Instances that are actually positive and are correctly predicted as positive.
- **False Positives (FP):** Instances that are actually negative but are incorrectly predicted as positive.
- **True Negatives (TN):** Instances that are actually negative and are correctly predicted as negative.
- **False Negatives (FN):** Instances that are actually positive but are incorrectly predicted as negative.



ACTUAL PREDICTED	POSITIVE	NEGATIVE
POSITIVE	TRUE POSITIVE (TP)	FALSE POSITIVE (FP)
NEGATIVE	FALSE NEGATIVE (FN)	TRUE NEGATIVE (TN)

Accuracy:

Accuracy is the most basic evaluation metric, representing the proportion of correct predictions out of the total predictions made. It is calculated as the ratio of the number of correctly classified instances to the total number of instances. However, accuracy alone may not provide a complete picture, especially when the classes are imbalanced.

Accuracy = $\frac{TP+TN}{TP\ +FP+\ TN+FN}$

Precision:

Precision measures the proportion of correctly predicted positive instances out of all predicted positive instances. It focuses on the accuracy of positive predictions. Precision is calculated as the ratio of true positives (correctly predicted positives) to the sum of true positives and false positives (incorrectly predicted positives).

Precision = $\frac{TP}{TP\ +\ FP}$

Recall or Sensitivity or True Positive Rate:

Recall measures the proportion of correctly predicted positive instances out of all actual positive instances. It focuses on the model's ability to find all positive instances. Recall is calculated as the ratio of true positives to the sum of true positives and false negatives (missed positives).

Recall (Sensitivity) = $\frac{TP}{TP\ +\ FN}$

F1 Score:

The F1 score combines precision and recall into a single metric, providing a balanced evaluation. It is the harmonic mean of precision and recall, calculated as 2 * (precision * recall) / (precision + recall). The F1 score is useful when the class distribution is imbalanced.

F1 Score = $\frac{2 * (Precision * Recall)}{(Precision + Recall)}$

Specificity or True Negative Rate:

Specificity measures the proportion of correctly predicted negative instances out of all actual negative instances. It focuses on the model's ability to identify negative instances. Specificity is calculated as the ratio of true negatives (correctly predicted negatives) to the sum of true negatives and false positives (incorrectly predicted negatives).

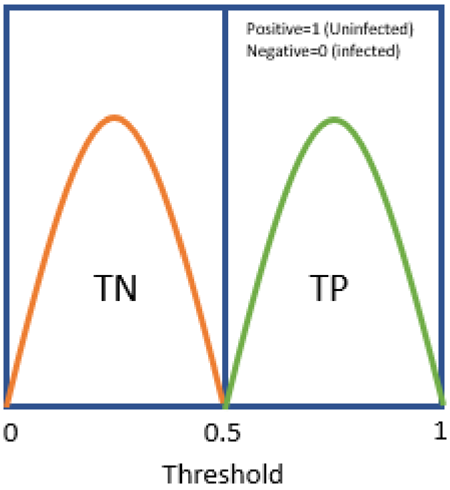
Specificity (True Negative Rate) = $\frac{TN}{TN+FP}$

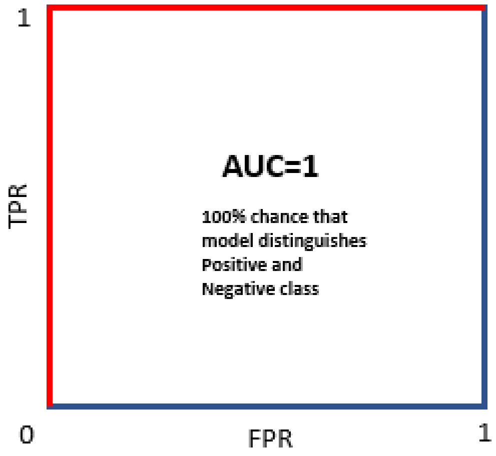
Area Under the Receiver Operating Characteristic Curve (AUC-ROC):

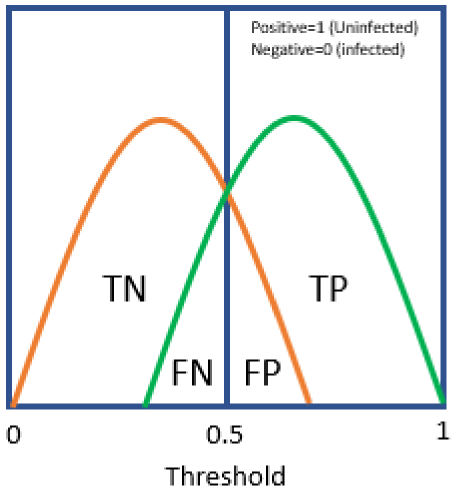
The ROC curve is a graphical representation of the model's performance at different classification thresholds. AUC-ROC represents the area under the ROC curve, which provides an aggregate measure of the model's discrimination ability. A higher AUC-ROC indicates better classification performance.

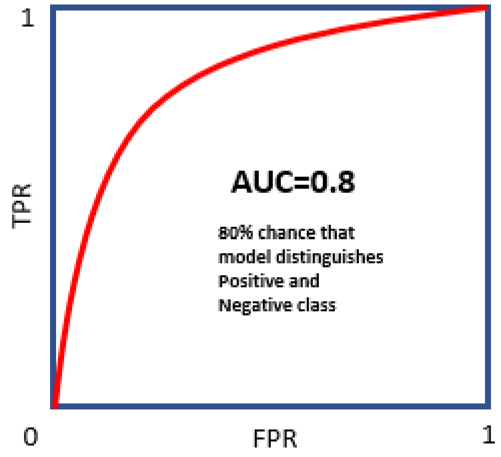
To find,

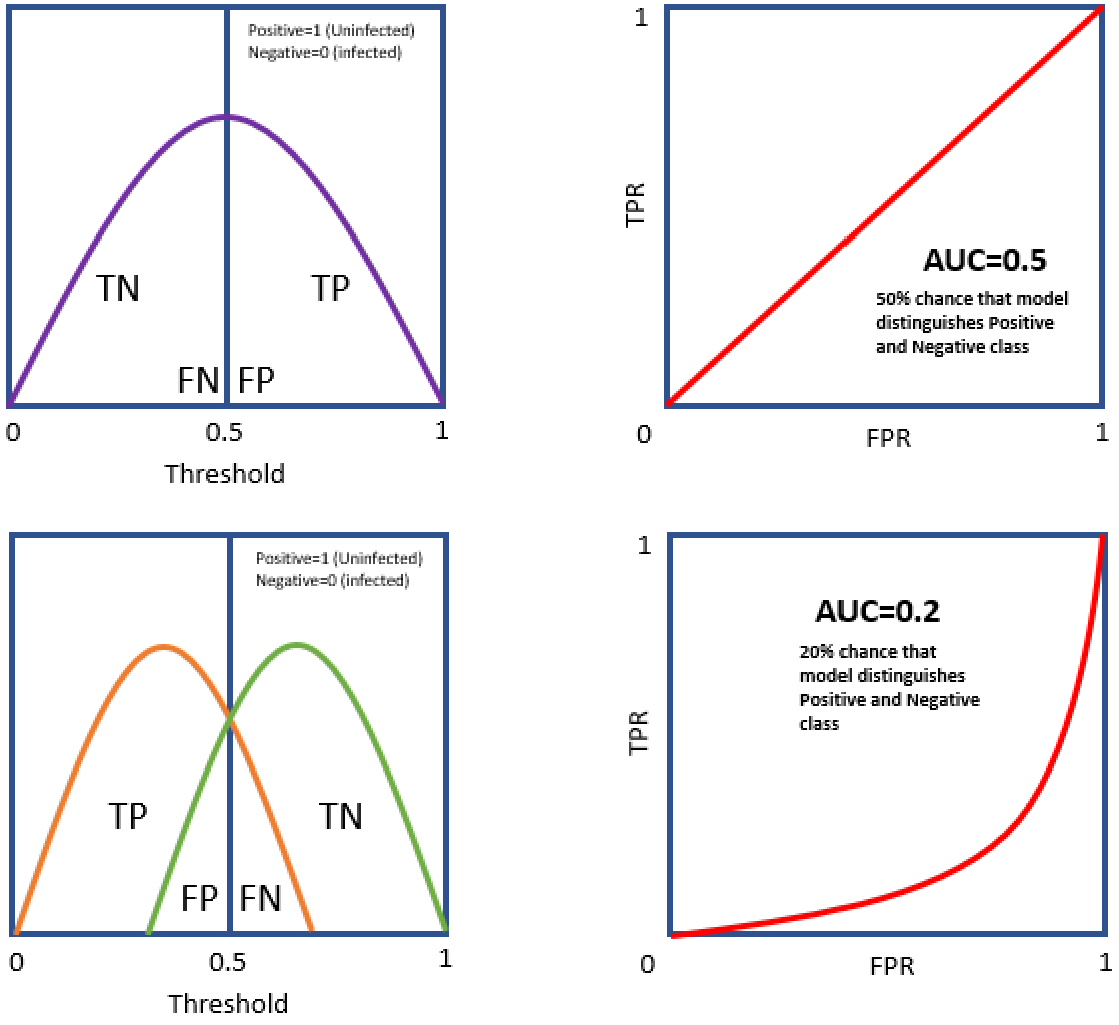
1. Sort the predicted probabilities or scores in descending order.
5. Calculate the True Positive Rate (TPR) and False Positive Rate (FPR) for each threshold. TPR is the ratio of true positives to the total number of actual positives, and FPR is the ratio of false positives to the total number of actual negatives.
6. Plot the points on a graph with TPR on the y-axis and FPR on the x-axis.
7. Calculate the area under the curve using a suitable numerical integration method, such as the trapezoidal rule or Simpson's rule.










Positive=1 (Uninfected)
Negative=0 (infected)
TN
TP
FN
FP
0
0.5
1
Threshold
1
TPR
AUC=0.5
50% chance that model distinguishes Positive and Negative class
0
FPR
1
Positive=1 (Uninfected)
Negative=0 (infected)
TP
TN
FP
FN
0
0.5
1
Threshold
1
TPR
AUC=0.2
20% chance that model distinguishes Positive and Negative class
0
FPR
1

Approach

Theoretical Approach:

1. **Define the problem:** Clearly articulate the problem you want to solve or the goal you want to achieve with ML. Determine if it's a classification, regression, clustering, or any other type of problem.

8. **Gather and explore the data:** Collect the relevant dataset for your problem domain. Explore the data to understand its structure, quality, and relationships. Perform descriptive statistics, visualization, and data pre-processing tasks such as handling missing values, outliers, and feature scaling.

9. **Split the dataset:** Divide the dataset into two or three parts, typically training, validation, and test sets. The training set is used to train the ML algorithm, the validation set helps in hyperparameter tuning, and the test set is used to evaluate the final model's performance.

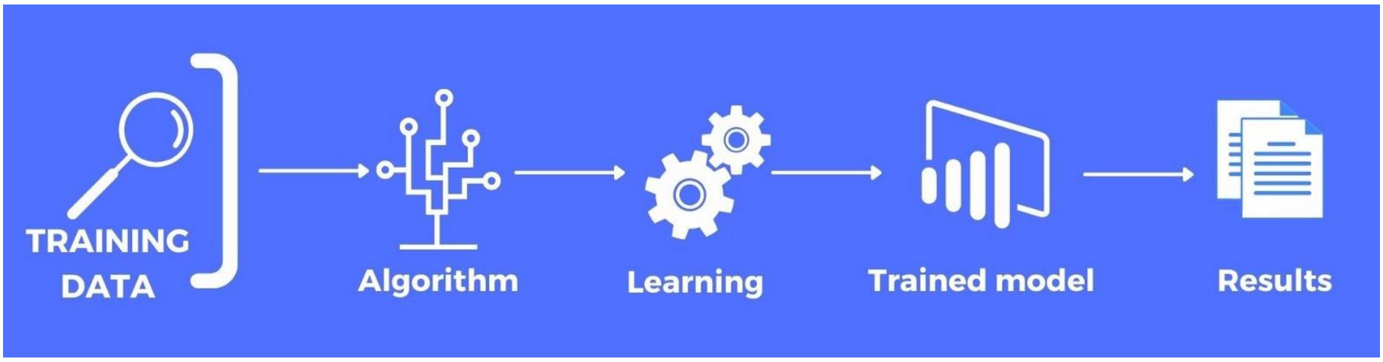
10. **Feature engineering:** Extract or create meaningful features from the dataset that can improve the ML model's performance. This might involve feature selection, dimensionality reduction techniques like Principal Component Analysis (PCA), or creating new features through transformations or domain knowledge.

11. **Select an appropriate algorithm:** Based on the problem type, dataset size, complexity, and other factors, choose a suitable ML algorithm. Consider algorithms such as decision trees, random forests, support vector machines, neural networks, or ensemble methods like gradient boosting or stacking.

12. **Train the model:** Feed the training dataset into the chosen algorithm and let it learn the underlying patterns and relationships. Adjust the algorithm's hyperparameters (e.g., learning rate, regularization strength) to optimize the model's performance. Use the validation set to fine-tune the hyperparameters through techniques like grid search or random search.

13. **Test the model:** Finally, evaluate the model's performance on the test set, which provides an unbiased assessment of its generalization capabilities. Ensure that the model's performance on the test set is consistent with the validation set.

14. **Evaluate the model:** Once the model is trained, assess its performance using appropriate evaluation metrics such as accuracy, precision, recall, F1-score, or mean squared error. Compare the model's performance on the validation set with different hyperparameter configurations to choose the best-performing model.
15. **Iterate and improve:** If the model's performance is not satisfactory, revisit previous steps. Explore alternative algorithms, perform more feature engineering, collect additional data, or refine the existing approach to improve the model's performance. Iterate until you achieve the desired results.
16. **Deploy the model:** Once you're satisfied with the model's performance, deploy it in a production environment to make predictions on new, unseen data. Monitor the model's performance over time and retrain or update it as needed.



Programming Approach:

When applying a machine learning (ML) algorithm to a dataset in Python, it's important to follow a systematic approach to ensure accurate results and efficient implementation. Here's a general outline of steps to follow:

1. **Import Required Libraries:** Importing the necessary libraries is the first step in any data analysis or machine learning project. Libraries like NumPy, Pandas, Scikit-learn, and Matplotlib provide various functions and tools for data manipulation, model training, and evaluation.
2. **Import Required Dataset(s):** Importing the dataset involves loading the data into a suitable data structure like a Pandas DataFrame or a NumPy array. This allows you to access and manipulate the data for further analysis.
3. **Check for any Null values:** Checking for null values is essential to ensure the quality and integrity of the dataset. Missing values can affect the performance of ML algorithms.

Handling missing values can involve either removing the rows or columns containing null values or filling them with appropriate values, such as the mean or median.

4. **Assign Depended and Independent variables:** In supervised learning tasks, the dataset is typically divided into two components: the independent variables (features) and the dependent variable (target variable). The independent variables are the inputs or predictors, while the dependent variable is the variable being predicted by the ML algorithm.
5. **Split the data into Training and Testing Datasets:** Splitting the data into training and testing datasets allows you to evaluate the performance of the ML model on unseen data. Typically, a certain percentage of the data (e.g., 80%) is used for training, while the remaining data is used for testing. This split helps assess how well the model generalizes to new, unseen data.
6. **Feature Scaling:** Feature scaling is the process of normalizing the numerical features in the dataset to a similar scale. Scaling is often required when the features have different magnitudes or units. Common scaling techniques include standardization (mean=0, standard deviation=1) and normalization (scaling values to a specific range, e.g., [0, 1]).
7. **Fit the ML Model:** Fitting the ML model involves training the chosen algorithm on the training dataset. The model learns patterns and relationships between the independent variables and the target variable to make predictions on new, unseen data. The model's parameters are adjusted during the training process to minimize the difference between predicted and actual values.
8. **Compute and Visualize Confusion Matrix:** The confusion matrix is a performance evaluation tool for classification problems. It provides a tabular representation of the model's predictions against the actual labels. It contains four metrics: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Visualizing the confusion matrix helps understand the model's performance in terms of correctly and incorrectly classified instances.
9. **Compute the Accuracy, Precision, Recall and F1 Score (Evaluation metrics):** Evaluation metrics provide quantitative measures to assess the model's performance. Common metrics include accuracy, precision, recall, and F1 score. Accuracy measures the overall

correctness of the model's predictions. Precision measures the model's ability to correctly identify positive instances. Recall (also called sensitivity or true positive rate) measures the model's ability to identify all positive instances. The F1 score combines precision and recall into a single metric that balances both metrics.

Remember that the specific techniques, algorithms, and evaluation metrics may vary depending on the problem type (classification, regression, clustering) and the characteristics of the dataset.

Introduction to Libraries used in Machine Learning Python

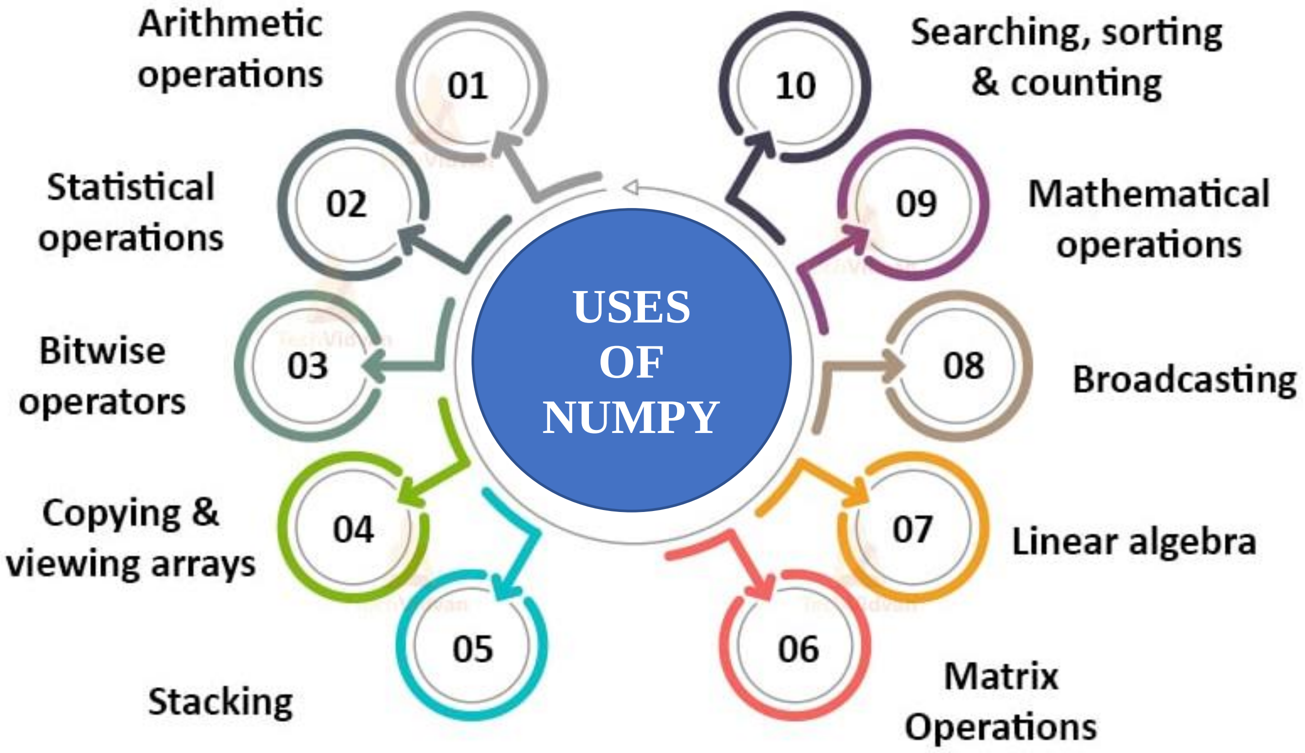
Machine learning is a rapidly evolving field, and there are several powerful libraries and frameworks available to assist with various aspects of the machine learning workflow. These libraries provide pre-built functions, algorithms, and tools that make it easier to develop, train, evaluate, and deploy machine learning models. Here are some of the most widely used and useful libraries in machine learning:

NumPy (Numerical Python):

NumPy is a fundamental library for numerical computing in Python. It provides a powerful N-dimensional array object and functions for manipulating arrays efficiently. NumPy is the foundation for many other libraries in the Python scientific ecosystem and is widely used for data manipulation and pre-processing in machine learning.

To import; ***import numpy as np***

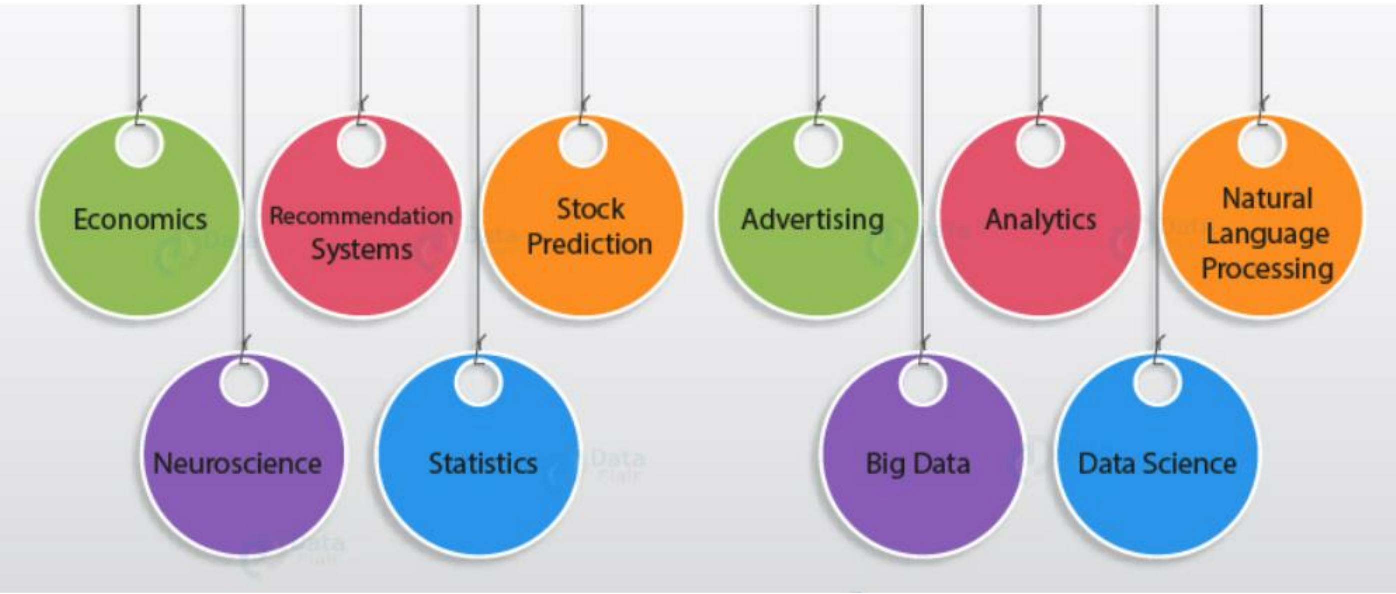
APPLICATIONS OF NUMPY:



Pandas:

Pandas is a popular library for data manipulation and analysis. It provides data structures such as DataFrames that make it easy to work with structured data. Pandas offers functions for data cleaning, transformation, and exploration, making it useful for data pre-processing tasks in machine learning.

To import; ***import pandas as pd***



Scikit-learn:

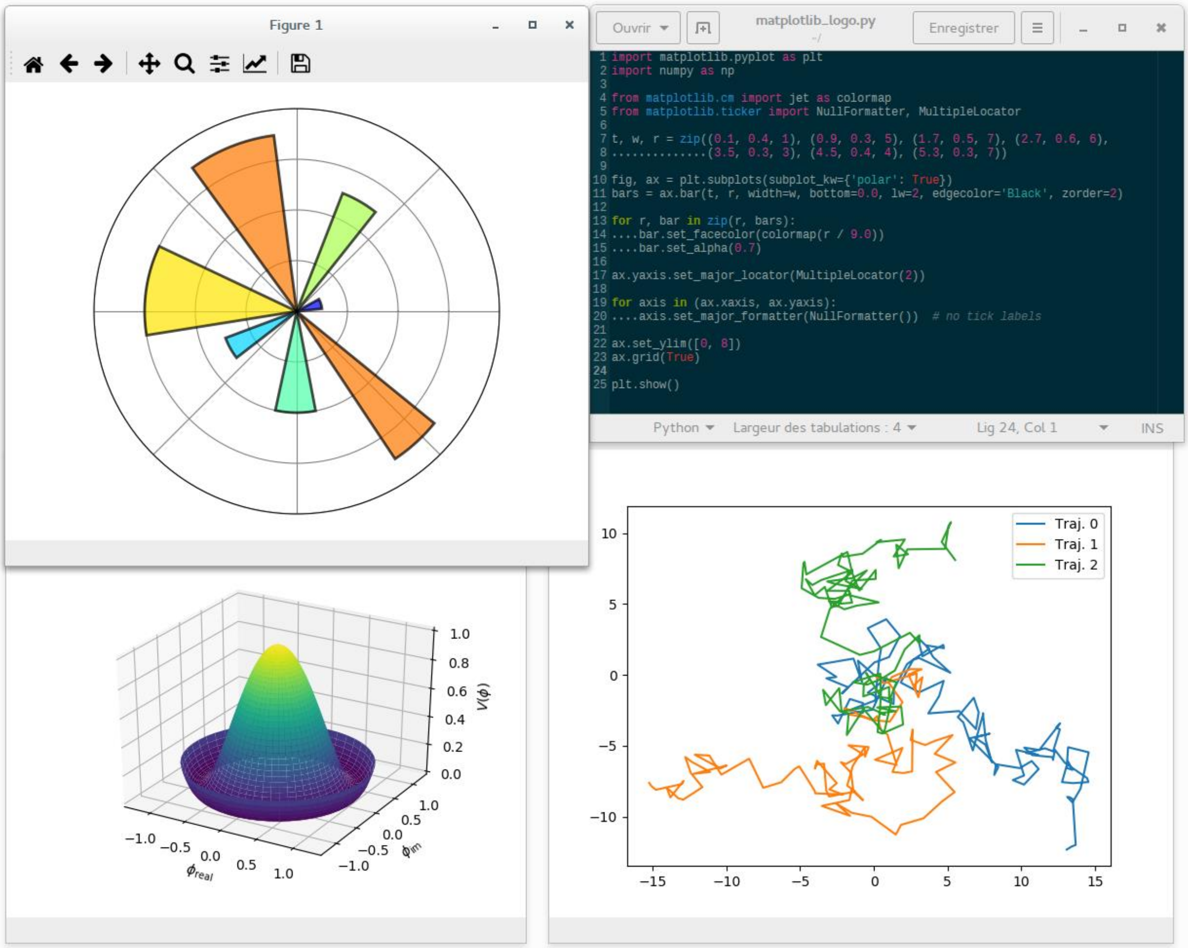
Scikit-learn is a comprehensive library for machine learning in Python. It provides a wide range of algorithms and tools for classification, regression, clustering, dimensionality reduction, and model evaluation. Scikit-learn is known for its simplicity and ease of use, making it an excellent choice for beginners.

To import; ***import sklearn***

Matplotlib:

Matplotlib is a widely used plotting library in Python. It provides a variety of functions to create high-quality visualizations, including line plots, scatter plots, bar plots, histograms, and more.

To import; ***import matplotlib.pyplot as plt***

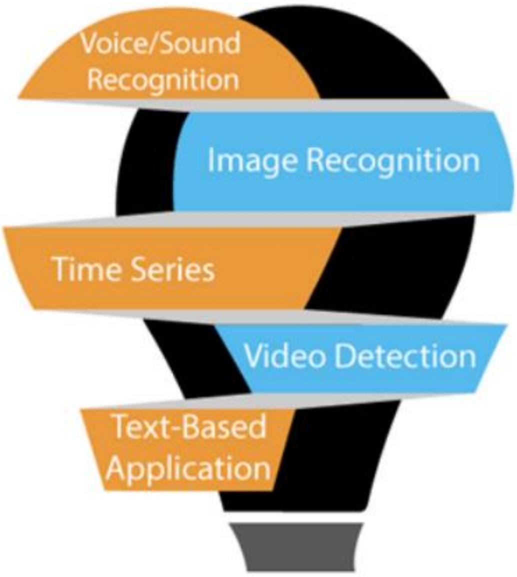


```
import matplotlib.pyplot as plt
import numpy as np

from matplotlib.cm import jet as colormap
from matplotlib.ticker import NullFormatter, MultipleLocator

t, w, r = zip((0.1, 0.4, 1), (0.9, 0.3, 5), (1.7, 0.5, 7), (2.7, 0.6, 6),
..............(3.5, 0.3, 3), (4.5, 0.4, 4), (5.3, 0.3, 7))

fig, ax = plt.subplots(subplot_kw={'polar': True})
bars = ax.bar(t, r, width=w, bottom=0.0, lw=2, edgecolor='Black', zorder=2)

for r, bar in zip(r, bars):
....bar.set_facecolor(colormap(r / 9.0))
....bar.set_alpha(0.7)

ax.yaxis.set_major_locator(MultipleLocator(2))

for axis in (ax.xaxis, ax.yaxis):
....axis.set_major_formatter(NullFormatter())  # no tick labels

ax.set_ylim([0, 8])
ax.grid(True)

plt.show()
```

TensorFlow:

TensorFlow is a powerful open-source library for numerical computation and machine learning developed by Google. It offers a flexible framework for building and deploying machine learning models, with a focus on deep learning. TensorFlow provides a high-level API called Keras, which simplifies the process of building neural networks.

To import; ***import tensorflow as tf***



Keras:

Keras is a high-level neural networks API written in Python. Initially developed as a user-friendly interface for building deep learning models on top of TensorFlow, it has since been integrated into TensorFlow's core library. Keras provides a simple and intuitive interface for designing and training neural networks.

To import; ***import keras***

PyTorch:

PyTorch is another popular open-source machine learning library that focuses on dynamic computation graphs. It offers a flexible and efficient framework for training deep learning models. PyTorch provides extensive support for GPU acceleration and is widely used in the research community.

To import; ***import torch***

XGBoost:

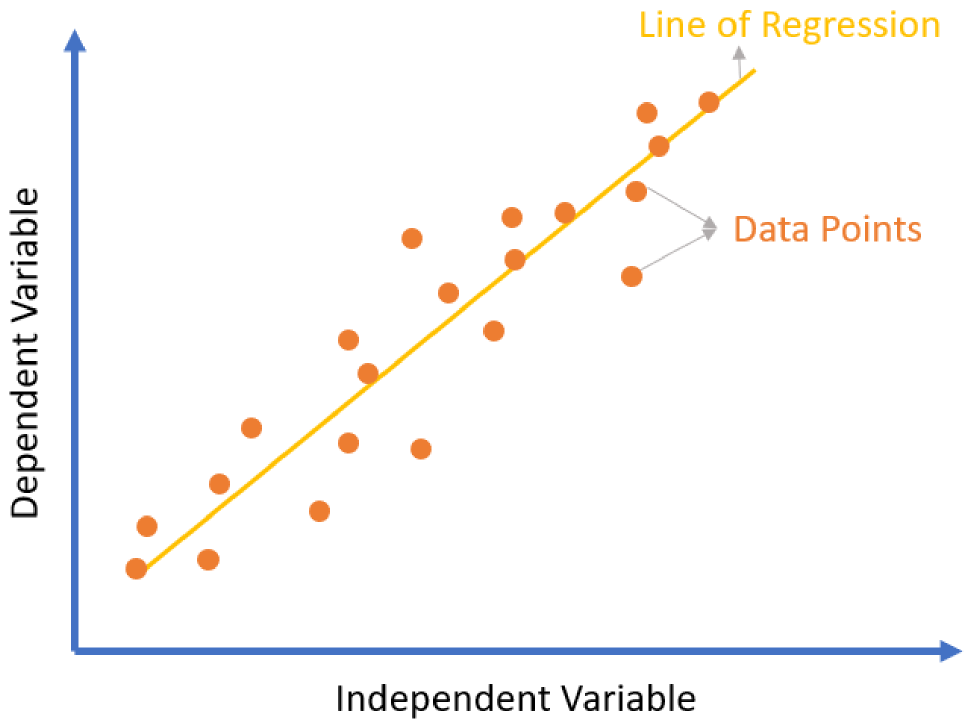
XGBoost is an optimized gradient boosting library that excels in handling tabular data and is widely used for classification and regression tasks. It provides an implementation of the gradient boosting algorithm, which combines multiple weak models to create a more accurate ensemble model.

To import; ***import xgboost as xgb***

1. Linear Regression

- ➢ Supervised Learning Model
- ➢ Mainly used for Regression tasks
- ➢ Suitable for predicting continuous target variables
- ➢ Line of Best Fit

Linear Regression is a popular and widely used supervised learning algorithm used for predicting continuous target variables based on one or more input features. It assumes a linear relationship between the input variables (features) and the output variable (target).



Assumptions of Linear Regression:

- **Linearity:** It assumes a linear relationship between the input features and the target variable.
- **Independence:** The input features should be independent of each other (no multicollinearity).
- **Homoscedasticity:** The residuals (the differences between the actual and predicted values) should have a constant variance across all levels of the input variables.
- **Normality:** The residuals should follow a normal distribution.

Simple Linear Regression:

Simple Linear Regression is the basic form of Linear Regression involving a single input feature (X) and a single target variable (y). The relationship is represented by the equation:

y = b0 + b1*X

where,

- y is the target variable.
- X is the input feature.
- b0 is the y-intercept (the value of y when X is zero).
- b1 is the slope (the change in y for a one-unit change in X).

The goal is to estimate the values of b0 and b1 that best fit the given data. This is typically done by minimizing the sum of squared errors (SSE) or by maximizing the likelihood function.

Multiple Linear Regression:

Multiple Linear Regression extends the simple linear regression to include multiple input features (X1, X2, ..., Xn) and a single target variable (y). The relationship is represented by the equation:

y = b0 + b1*X1 + b2*X2 + ... + bn*Xn

where:

- y is the target variable.
- X1, X2, ..., Xn are the input features.
- b0 is the y-intercept.
- b1, b2, ..., bn are the slopes associated with each input feature.

The goal remains the same: to estimate the values of b0, b1, b2, ..., bn that best fit the given data.

Model Evaluation:

To assess the performance of a linear regression model, several evaluation metrics are commonly used:

- **Mean Squared Error (MSE):** It measures the average squared difference between the predicted and actual values. A lower MSE indicates better model performance.
- **Root Mean Squared Error (RMSE):** It is the square root of the MSE and provides the measure of the average prediction error in the same units as the target variable.
- **R-squared (R2) Score:** It represents the proportion of the variance in the target variable that can be explained by the model. It ranges from 0 to 1, with 1 indicating a perfect fit.
- **Adjusted R-squared Score:** It adjusts the R-squared score by considering the number of input features and the sample size. It penalizes the addition of irrelevant features.

Limitations of Linear Regression:

Linear Regression has certain limitations that should be considered:

- **Linearity Assumption:** Linear Regression assumes a linear relationship between the input features and the target variable. If the relationship is non-linear, Linear Regression may not provide accurate predictions.
- **Sensitive to Outliers:** Linear Regression is sensitive to outliers, as they can significantly impact the estimated coefficients and the model's performance.
- **Assumptions Violation:** If the assumptions of Linear Regression (linearity, independence, homoscedasticity, normality) are violated, the model's performance may be affected.
- **Multicollinearity:** Linear Regression assumes independence between input features. When features are highly correlated (multicollinearity), it can lead to unstable and unreliable coefficient estimates.
- **Limited to Linear Relationships:** Linear Regression is not suitable for capturing complex non-linear relationships between features and the target variable.

Applications of Linear Regression:

- **Economics and Finance:** Linear Regression is extensively used in economic analysis, financial modeling, and forecasting. It can help analyze the relationship between economic variables, predict stock prices, estimate demand and supply, evaluate the impact of policies, and assess risk.

- **Marketing and Sales:** Linear Regression is employed in market research and sales forecasting. It can assist in understanding the factors influencing consumer behavior, predicting product demand, optimizing pricing strategies, and measuring the effectiveness of marketing campaigns.
- **Social Sciences:** Linear Regression is used in social science research to analyze relationships between variables. It can help examine factors affecting education outcomes, assess social and economic disparities, study population trends, and analyze survey data.
- **Healthcare:** Linear Regression is applied in healthcare for various purposes, including analyzing the impact of medical treatments, predicting patient outcomes, modeling disease progression, and estimating healthcare costs.
- **Real Estate:** Linear Regression can be utilized in real estate for property price prediction, rental price estimation, assessing market trends, and evaluating the impact of location and property characteristics.
- **Environmental Science:** Linear Regression is used in environmental studies to analyze the relationships between environmental variables, predict pollution levels, model climate change, and assess the impact of human activities on ecosystems.
- **Engineering and Manufacturing:** Linear Regression is employed in engineering and manufacturing for quality control, process optimization, predicting equipment failure, and optimizing production efficiency.

Implementing Linear Regression using Python

Dataset Required:

https://drive.google.com/file/d/1uesxH_CQprom9HqwhspvoesUyHo4KA7h/view?usp=sharing

Importing Required Libraries:

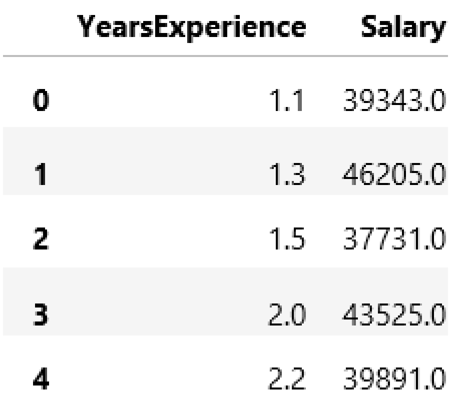
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
```

Importing (Reading) Datasets:

```
data=pd.read_csv('/content/Salary_Data.csv')
```

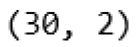
Exploring Dataset:

```
data.head()
```



	YearsExperience	Salary
0	1.1	39343.0
1	1.3	46205.0
2	1.5	37731.0
3	2.0	43525.0
4	2.2	39891.0

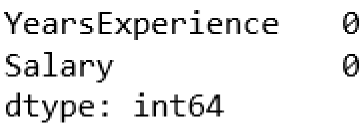
```
data.shape
```



```
(30, 2)
```

Checking for any null values in dataset:

```
data.isnull().sum()
#Checking for any Null values in the imported Datasets
```



```
YearsExperience    0
Salary             0
dtype: int64
```

Assigning Dependent and Independent variables:

```
x=data.iloc[:,:1].values
y=data.iloc[:, 1:2].values
```

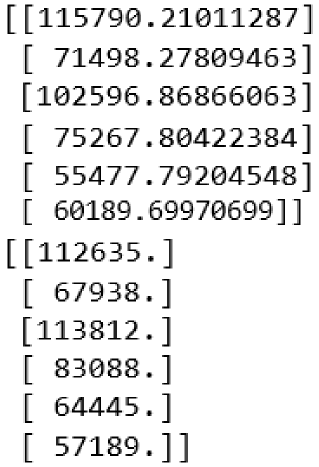
Splitting the dataset into Training and Testing Dataset:

```
x_train, x_test, y_train, y_test = train_test_split(x,y,
      test_size=0.2, random_state = 42)
```

Fitting the Model (Linear Regression):

```
model=LinearRegression()
model.fit(x_train, y_train)
y_pred=model.predict(x_test)
```

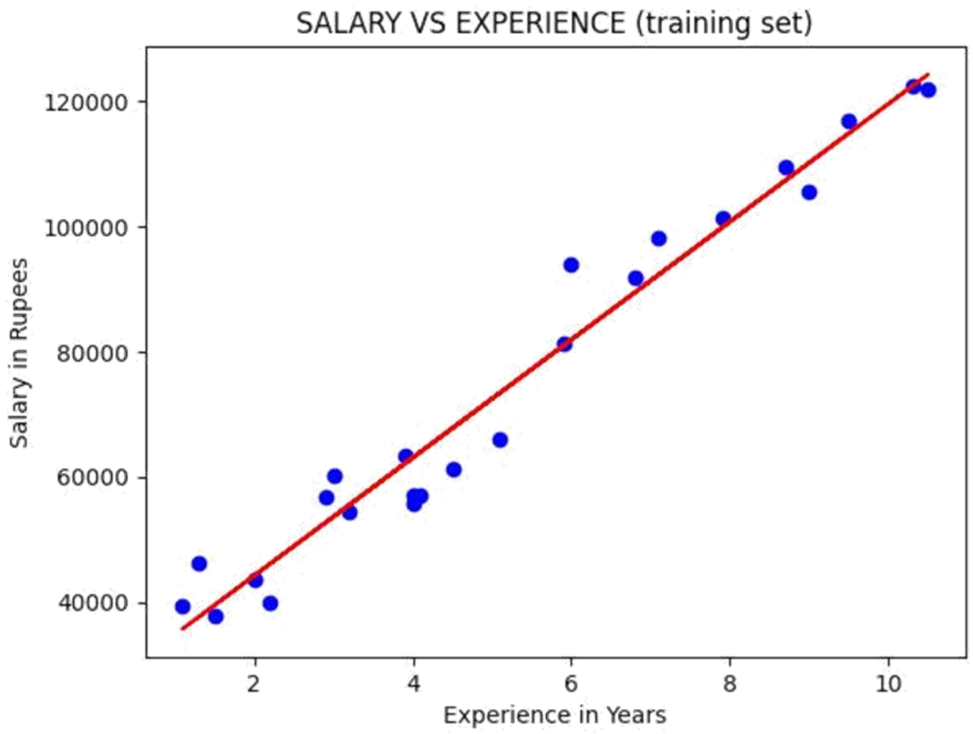
```
print(y_pred)
print(y_test)
```



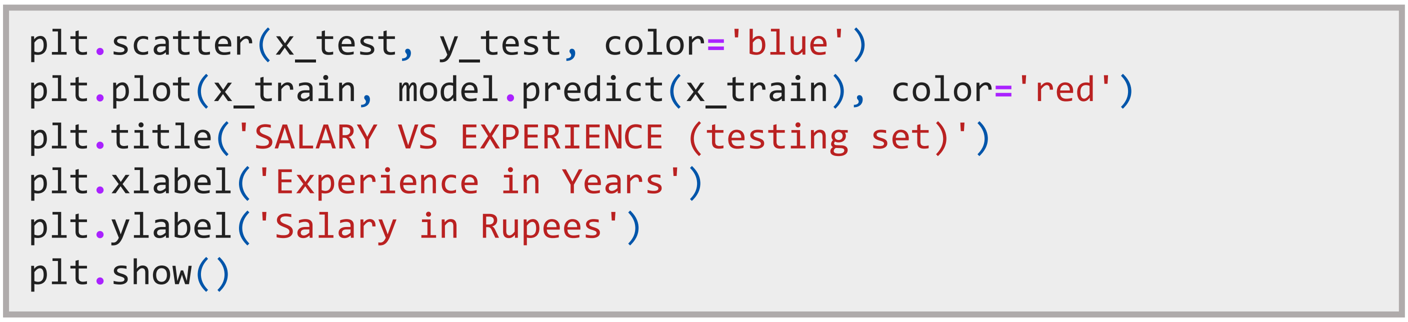
```
[[115790.21011287]
 [ 71498.27809463]
 [102596.86866063]
 [ 75267.80422384]
 [ 55477.79204548]
 [ 60189.69970699]]
[[112635.]
 [ 67938.]
 [113812.]
 [ 83088.]
 [ 64445.]
 [ 57189.]]
```

Plot for Training dataset

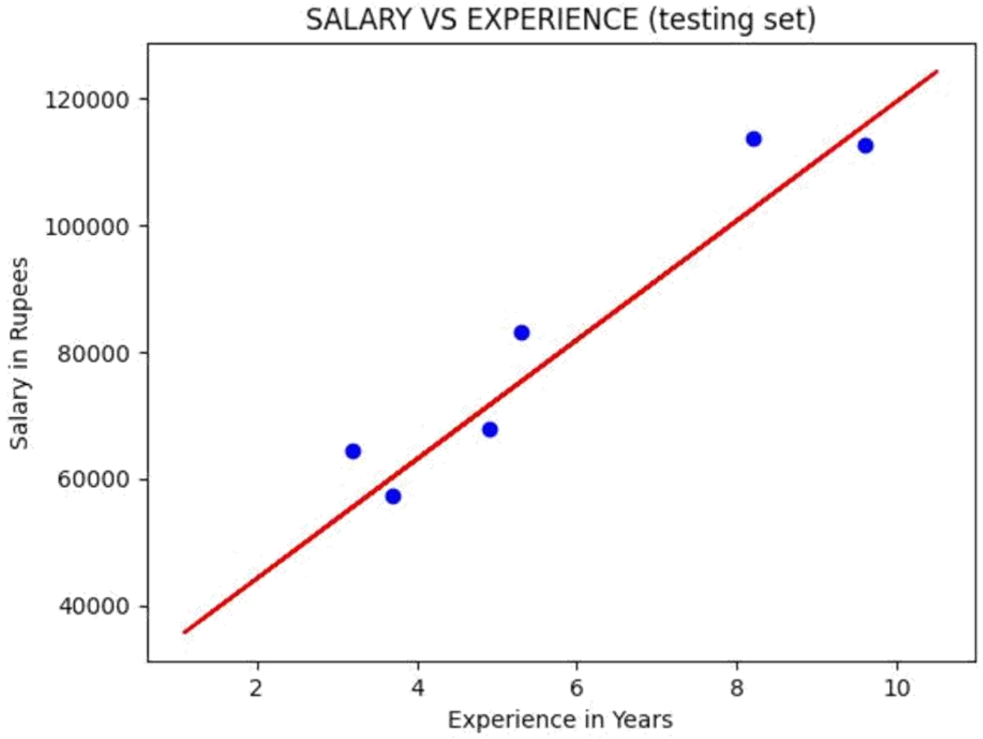
```
plt.scatter(x_train, y_train, color='blue')
plt.plot(x_train, model.predict(x_train), color='red')
plt.title('SALARY VS EXPERIENCE (training set)')
plt.xlabel('Experience in Years')
plt.ylabel('Salary in Rupees')
plt.show()
```



Plot for Testing dataset



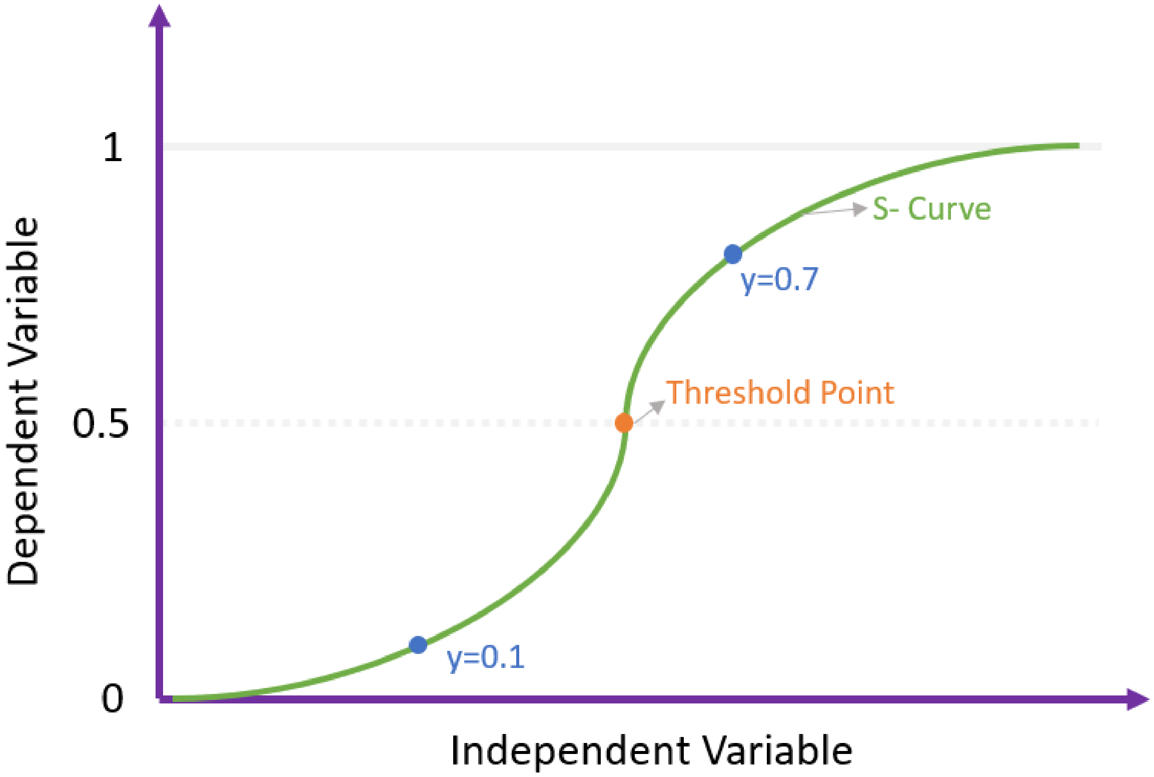
```
plt.scatter(x_test, y_test, color='blue')
plt.plot(x_train, model.predict(x_train), color='red')
plt.title('SALARY VS EXPERIENCE (testing set)')
plt.xlabel('Experience in Years')
plt.ylabel('Salary in Rupees')
plt.show()
```



2. Logistic Regression

- Supervised Learning Model
- Primarily used for binary classification problems and Regression
- Linear regression + Sigmoid Function

Logistic regression is a statistical model used for binary classification problems. It is an extension of linear regression that predicts the probability of an input belonging to a specific class. Unlike linear regression, which predicts continuous values, logistic regression is designed to handle discrete outcomes.



The fundamental concept behind logistic regression is the logistic function, also known as the sigmoid function. The logistic function maps any real number to a value between 0 and 1. It takes the form:

sigmoid(z) = $\frac{1}{(1 + exp(-z))}$

In logistic regression, the model applies this sigmoid function to a linear combination of the input features to obtain a value between 0 and 1. This value represents the estimated probability of the input belonging to a particular class.

To learn the parameters of the logistic regression model, it uses a technique called maximum likelihood estimation. The objective is to find the optimal set of coefficients that maximizes the likelihood of observing the labeled data. This process involves minimizing a cost function, often referred to as the cross-entropy loss, which measures the dissimilarity between the predicted probabilities and the true class labels.

Once the model is trained, it can make predictions on new, unseen data by calculating the probability of the input belonging to the positive class (class 1) based on the learned coefficients and feature values. By applying a chosen threshold (commonly 0.5), the model classifies the input into one of the two classes: positive or negative.

Logistic regression has several advantages. Firstly, it is a relatively simple and interpretable model. The coefficients can be easily interpreted as the influence of each feature on the probability of the outcome. This makes logistic regression useful for understanding the relationship between predictors and the response variable.

Additionally, logistic regression is computationally efficient and can handle large datasets. It requires fewer computational resources compared to more complex models like neural networks. Moreover, logistic regression provides probabilistic outputs, allowing for a better understanding of the uncertainty associated with each prediction.

However, logistic regression also has some limitations. It assumes a linear relationship between the input features and the log-odds of the outcome. If the relationship is non-linear, logistic regression may not capture it effectively. In such cases, feature engineering or more advanced techniques may be necessary.

Logistic regression is also sensitive to outliers. Outliers can disproportionately affect the estimated coefficients, leading to biased predictions. Thus, it is important to preprocess the data and handle outliers appropriately.

Logistic regression finds application in various domains. It is commonly used in areas such as spam detection, fraud detection, disease diagnosis, sentiment analysis, and churn prediction. Its simplicity, interpretability, and efficiency make it a popular choice when transparency and explainability are important.

Advantages of Logistic Regression:

- **Simplicity:** Logistic regression is a relatively simple and interpretable model. It is easy to understand and implement, making it a good choice when transparency and explainability are important.
- **Efficiency:** Logistic regression can be trained efficiently even on large datasets. It has low computational requirements, making it computationally inexpensive compared to more complex models.
- **Probability estimation:** Logistic regression provides probabilistic outputs, allowing for a better understanding of the uncertainty associated with each prediction. This can be useful in decision-making processes.

Limitations of Logistic Regression:

- **Linearity assumption:** Logistic regression assumes a linear relationship between the input features and the log-odds of the outcome. If the relationship is non-linear, logistic regression may not perform well and may require feature engineering or more advanced techniques.
- **Limited complexity:** Logistic regression is a linear model and cannot capture complex relationships or interactions between features as effectively as non-linear models like decision trees or neural networks.
- **Sensitivity to outliers:** Logistic regression can be sensitive to outliers, as it tries to minimize the overall error. Outliers can disproportionately affect the estimated coefficients and, consequently, the predictions.

Applications of Logistic Regression:

- **Spam Detection:** Logistic regression can be used to identify spam emails by analyzing various features such as the email content, sender information, and subject line. It classifies emails as either spam or non-spam based on learned patterns from labeled training data.
- **Fraud Detection:** Logistic regression is utilized in fraud detection systems to identify fraudulent transactions or activities. By considering factors like transaction amounts,

locations, and user behavior patterns, the model can predict the likelihood of a transaction being fraudulent.

- **Disease Diagnosis:** Logistic regression is employed in medical research and healthcare to predict the presence of certain diseases or conditions. By considering patient characteristics, symptoms, and diagnostic test results, the model can assist in diagnosing diseases such as cancer, diabetes, or heart disease.
- **Sentiment Analysis:** Logistic regression is used in sentiment analysis to determine the sentiment or opinion expressed in textual data. It can classify text as positive or negative based on the presence of certain words, sentiment indicators, or linguistic patterns. This application is valuable in social media monitoring, brand reputation management, and customer feedback analysis.
- **Market Segmentation:** Logistic regression is used in market research and customer segmentation to divide a population into distinct groups based on their characteristics, preferences, or behaviors. By analyzing demographic data, purchasing patterns, or survey responses, the model can identify segments with similar traits for targeted marketing strategies.

Implementing Logistic Regression using Python

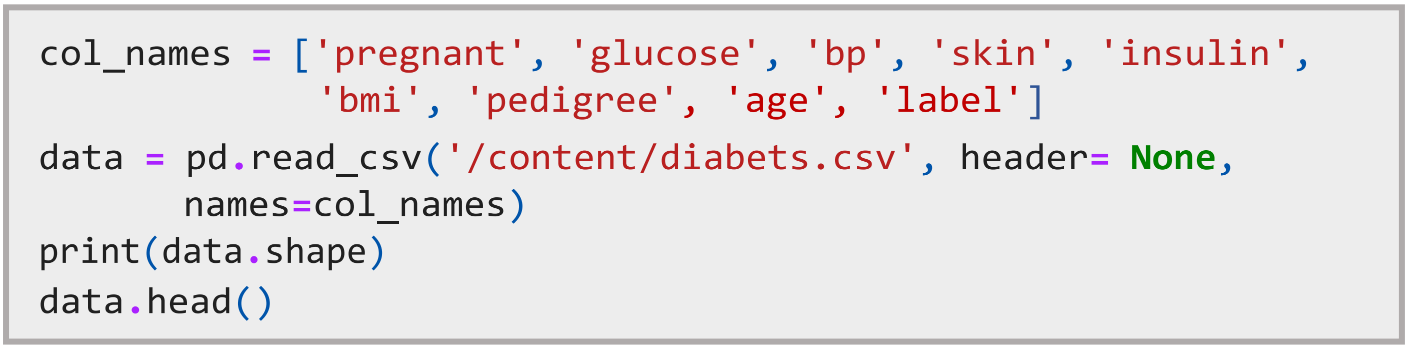
Dataset Required

https://drive.google.com/file/d/1V6yFU3nDdx9R56yOzy6GxHPq-Dav2A4K/view?usp=share_link

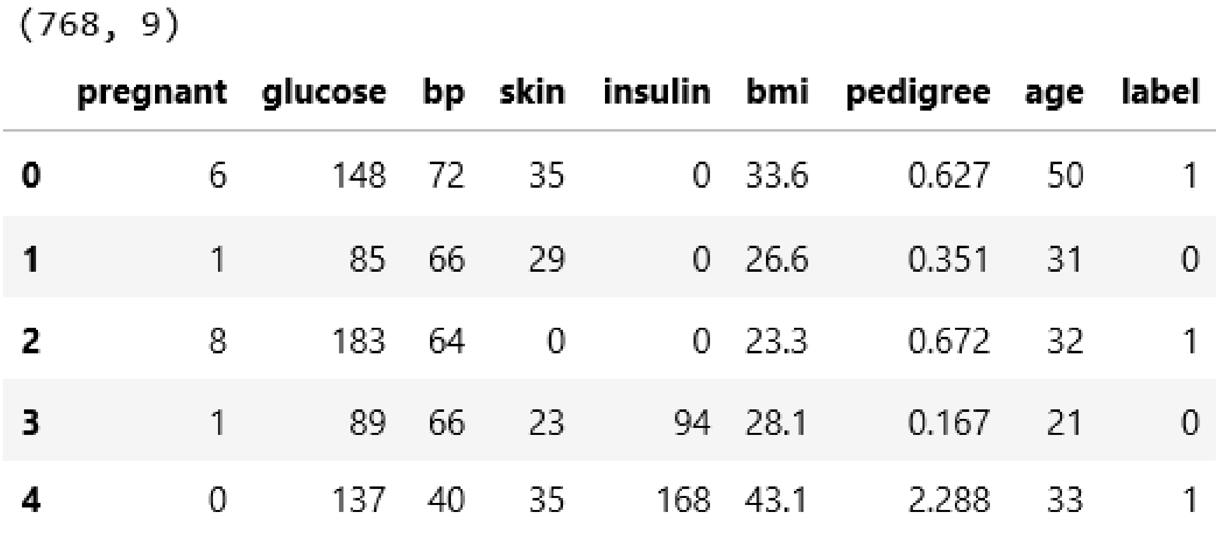
Importing Required Libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
import seaborn as sn
```

Importing (Reading) Datasets



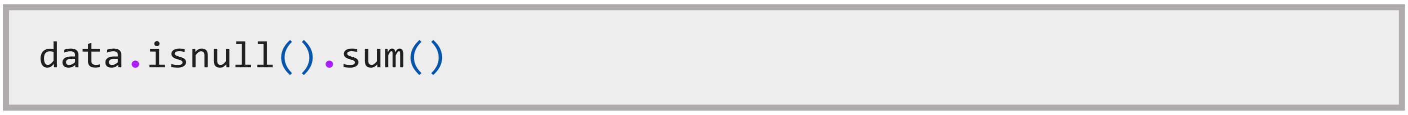
```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin',
             'bmi', 'pedigree', 'age', 'label']
data = pd.read_csv('/content/diabets.csv', header= None,
       names=col_names)
print(data.shape)
data.head()
```



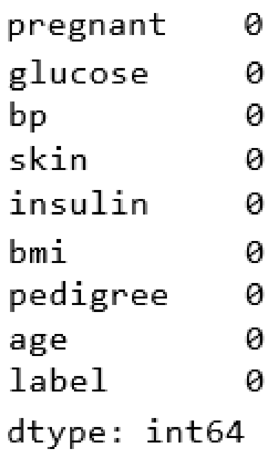
```
(768, 9)
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Checking for any null values in dataset



```
data.isnull().sum()
```



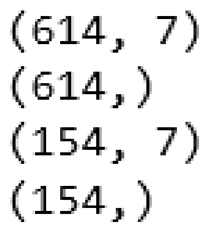
```
pregnant    0
glucose     0
bp          0
skin        0
insulin     0
bmi         0
pedigree    0
age         0
label       0
dtype: int64
```

Assigning dependent and independent variables

```
feature_cols = ['pregnant','insulin', 'bmi',
                 'age','glucose','bp', 'pedigree']
x=data[feature_cols]
y=data.label
```

splitting the dataset into Training and Testing Dataset

```
x_train, x_test, y_train, y_test = train_test_split(x,y,
    test_size=0.2, random_state=5)
display(x_train.shape, y_train.shape, x_test.shape,
    y_test.shape)
```



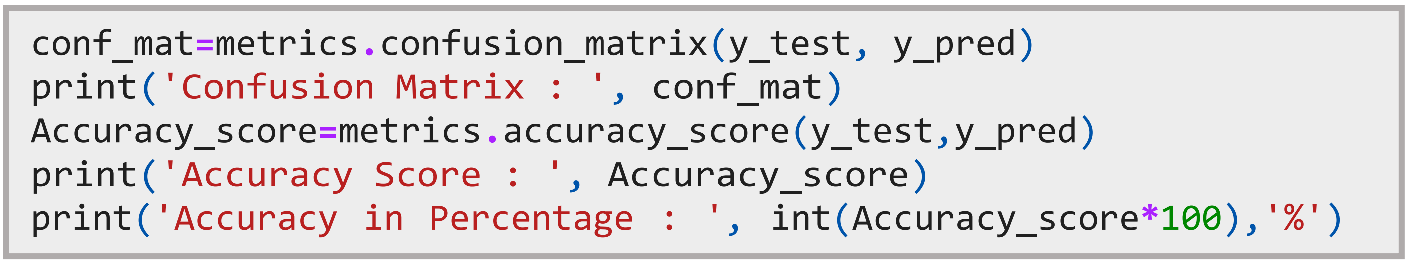
```
(614, 7)
(614,)
(154, 7)
(154,)
```

Fitting the Model (Logistic Regression)

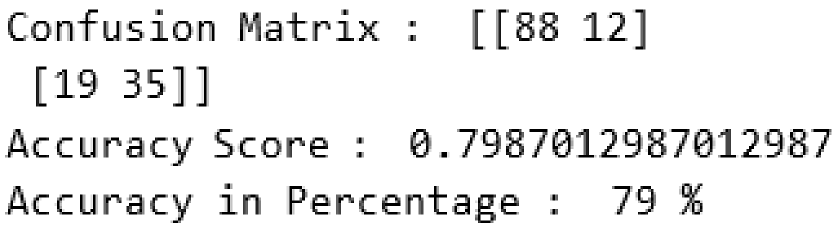
```
model= LogisticRegression(solver='lbfgs', max_iter=1000)
```

```
model.fit(x_train, y_train)
y_pred=model.predict(x_test)
```

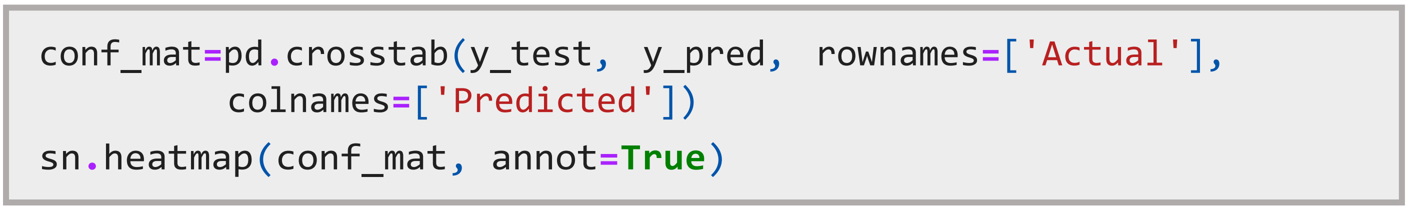
Evaluation Metrics



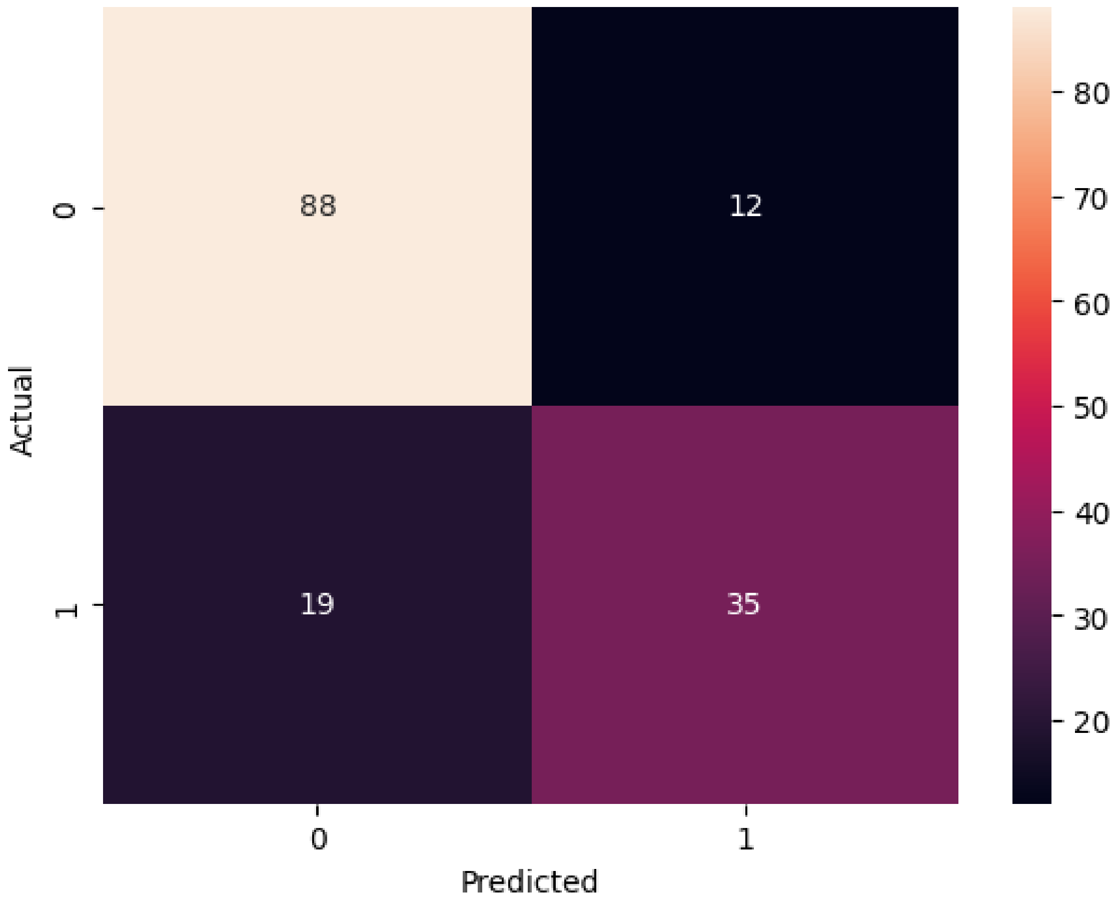
```
conf_mat=metrics.confusion_matrix(y_test, y_pred)
print('Confusion Matrix : ', conf_mat)
Accuracy_score=metrics.accuracy_score(y_test,y_pred)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ', int(Accuracy_score*100),'%')
```



```
Confusion Matrix :  [[88 12]
 [19 35]]
Accuracy Score :  0.7987012987012987
Accuracy in Percentage :  79 %
```



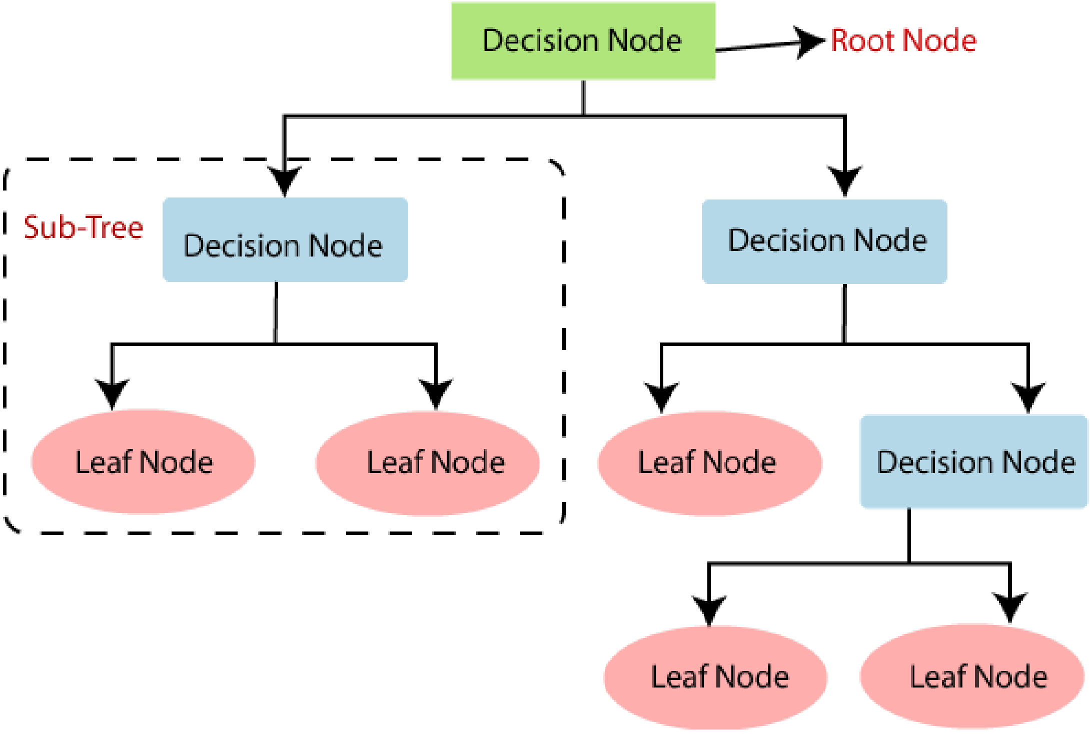
```
conf_mat=pd.crosstab(y_test, y_pred, rownames=['Actual'],
        colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True)
```



3. Decision Tree Classifier

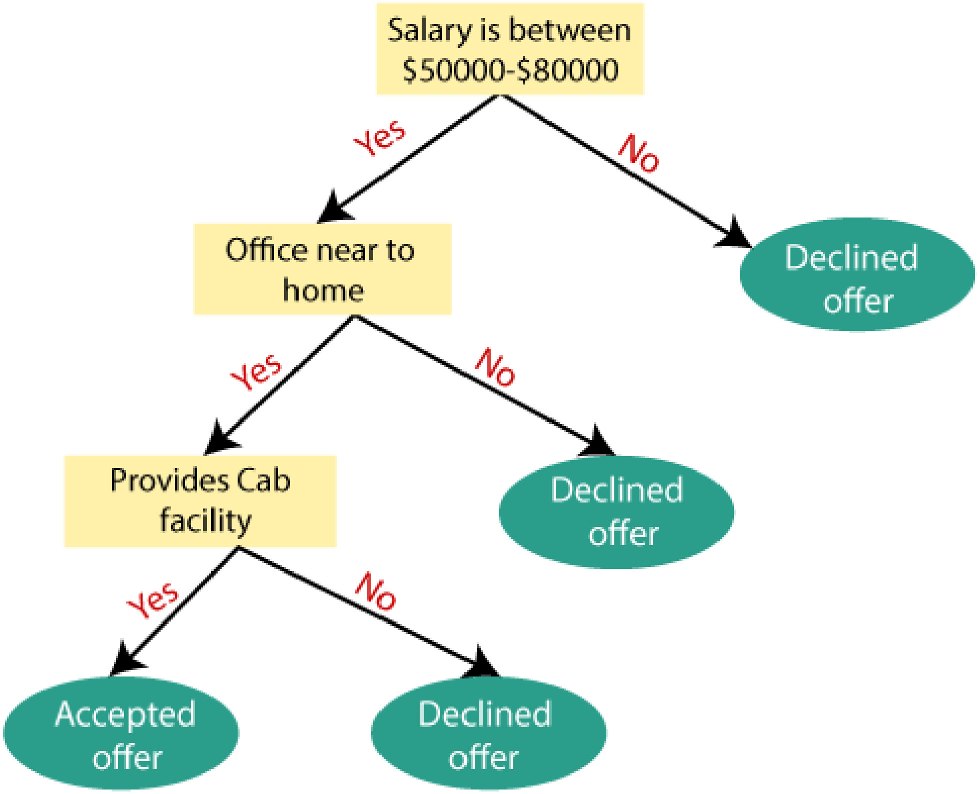
- Supervised Learning Model
- Tree structure Model

A decision tree classifier is a supervised machine learning algorithm that uses a tree-like structure to make predictions or classify input data. It recursively partitions the input space based on the features to create a tree of decision nodes and leaf nodes. Each decision node represents a feature and a threshold, while each leaf node represents a class label or a prediction.



The decision tree classifier operates by recursively splitting the input data based on the values of different features. It partitions the data into subsets at each decision node based on the selected feature and its threshold value. This process continues until the algorithm reaches a stopping criterion, such as reaching a maximum depth, a minimum number of samples, or when all samples in a subset belong to the same class.

The splitting process aims to maximize the homogeneity or purity of the subsets. Various splitting criteria can be used, with the most common being Gini impurity and entropy. Gini impurity measures the probability of misclassifying a randomly chosen sample if it were labeled randomly according to the distribution of classes in the subset. Entropy, on the other hand, measures the level of impurity or randomness in the subset.



Advantages of Decision Tree Classifier:

- **Interpretability:** Decision trees are easy to understand and interpret. The structure of the tree allows for transparent decision-making, as each path from the root to a leaf represents a set of rules that lead to a particular prediction or classification.
- **Handling Non-linearity:** Decision trees can handle non-linear relationships between features and the target variable without requiring complex transformations. They can capture interactions and non-linear decision boundaries by recursively splitting the data based on the features' values.
- **Feature Importance:** Decision trees provide a measure of feature importance by evaluating the influence of each feature in the tree structure. This information can be valuable for feature selection and understanding the underlying factors driving the predictions.
- **Handling Missing Values:** Decision trees can handle missing values in the dataset. They can evaluate the available features and select the optimal split based on the available data.

Limitations of Decision Tree Classifier:

- **Overfitting:** Decision trees have a tendency to overfit the training data, especially when the tree becomes deep and complex. Overfitting occurs when the tree captures noise or irrelevant patterns in the training data, leading to poor generalization on unseen data.
- **Instability:** Decision trees can be sensitive to small changes in the training data, leading to different tree structures and predictions. This instability can be reduced by using ensemble methods like random forests or boosting.
- **Lack of Smoothness:** Decision trees produce piecewise constant predictions, meaning they create boundaries between regions with different predictions. This lack of smoothness may not be suitable for problems where a smooth decision boundary is desired.

Applications of Decision Tree Classifier:

- **Credit Scoring:** Decision trees are commonly used in credit scoring to assess the creditworthiness of individuals or businesses. By considering factors such as income, credit history, and debt-to-income ratio, decision trees can predict the likelihood of a borrower defaulting on a loan.
- **Customer Churn Prediction:** Decision trees can predict customer churn by analyzing factors like customer demographics, purchase behavior, and service usage. This helps businesses identify customers at risk of churning and take proactive measures to retain them.
- **Disease Diagnosis:** Decision trees are employed in medical diagnosis to predict the presence of certain diseases or conditions. By considering symptoms, patient characteristics, and medical test results, decision trees can assist in diagnosing diseases and recommending appropriate treatments.
- **Fraud Detection:** Decision trees are used in fraud detection systems to identify fraudulent transactions or activities. By analyzing transaction patterns, user behavior, and other relevant features, decision trees can flag suspicious activities for further investigation.

Implementing Decision Tree Classifier using Python

Dataset Required

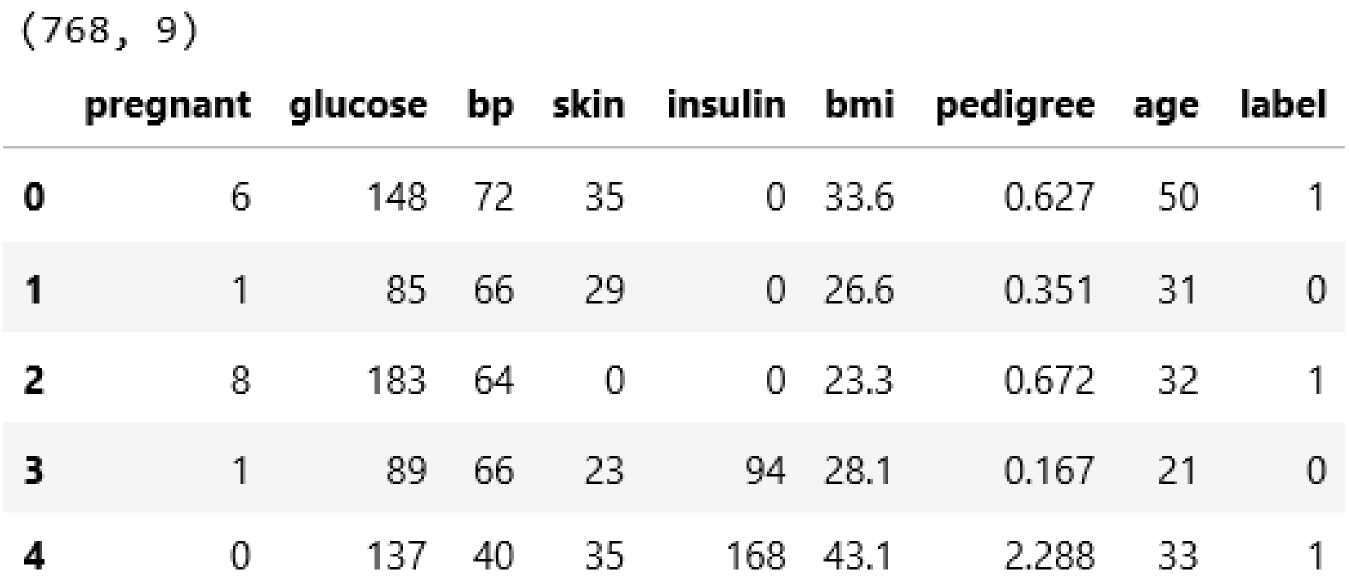
https://drive.google.com/file/d/1V6yFU3nDdx9R56yOzy6GxHPq-Dav2A4K/view?usp=share_link

Importing Required Libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
import seaborn as sn
```

Importing (Reading) Datasets

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin',
             'bmi', 'pedigree', 'age', 'label']
data = pd.read_csv('/content/diabets.csv', header= None,
       names=col_names)
print(data.shape)
data.head()
```



```
(768, 9)
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Checking for any null values in dataset

```
data.isnull().sum()
```

```
pregnant    0
glucose     0
bp          0
skin        0
insulin     0
bmi         0
pedigree    0
age         0
label       0
dtype: int64
```

Assigning dependent and independent variables

```
feature_cols = ['pregnant','insulin', 'bmi',
                'age','glucose','bp', 'pedigree']
x=data[feature_cols]
y=data.label
```

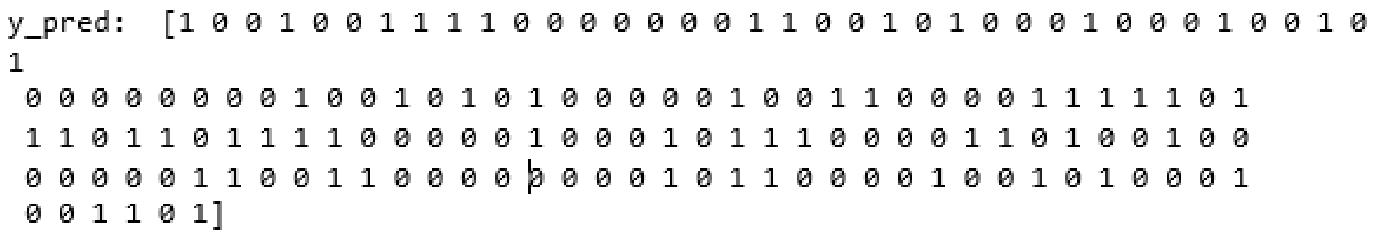
splitting the dataset into Training and Testing Dataset

```
x_train, x_test, y_train, y_test = train_test_split(x,y,
     test_size=0.2, random_state=5)
display(x_train.shape, y_train.shape, x_test.shape,
     y_test.shape)
```

```
(614, 7)
(614,)
(154, 7)
(154,)
```

Fitting the Model (Decision Tree Classifier)

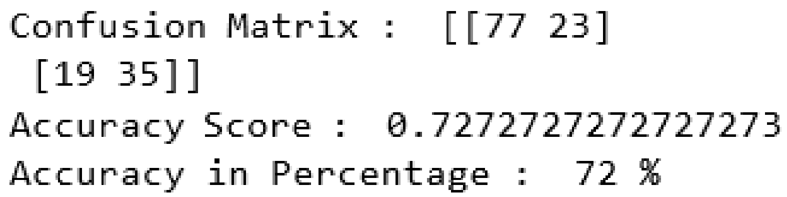
```
model=  DecisionTreeClassifier(criterion='entropy',random_state=5)
model.fit(x_train, y_train)
y_pred=model.predict(x_test)
print('y_pred: ', y_pred)
```



```
y_pred:  [1 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 1 0
1
 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1 1 1 1 1 0 1
 1 1 0 1 1 0 1 1 1 1 0 0 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 1 1 0 1 0 0 1 0 0
 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 1 0 1 0 0 0 1
 0 0 1 1 0 1]
```

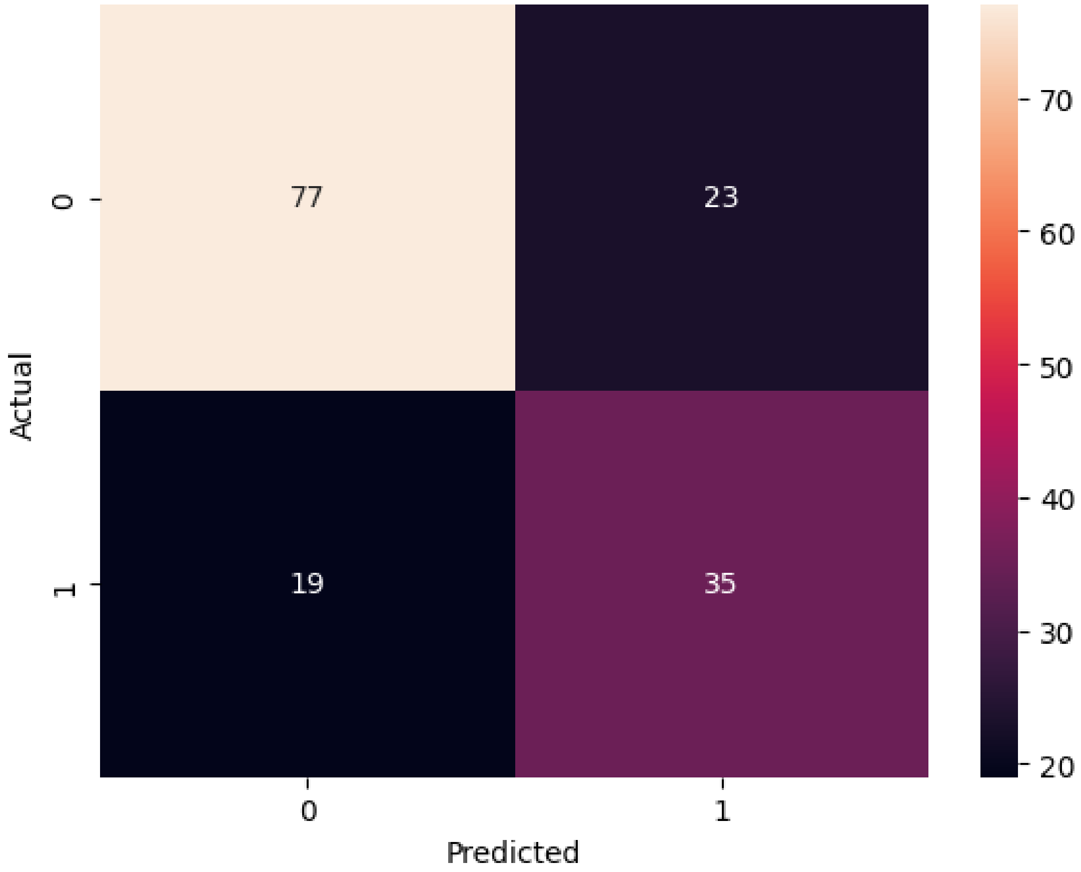
Evaluation Metrics

```
conf_mat=metrics.confusion_matrix(y_test, y_pred)
print('Confusion Matrix : ', conf_mat)
Accuracy_score=metrics.accuracy_score(y_test, y_pred)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ', int(Accuracy_score*100),'%')
```



```
Confusion Matrix :  [[77 23]
 [19 35]]
Accuracy Score :  0.7272727272727273
Accuracy in Percentage :  72 %
```

```
conf_mat=pd.crosstab(y_test, y_pred, rownames=['Actual'],
    colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True)
```

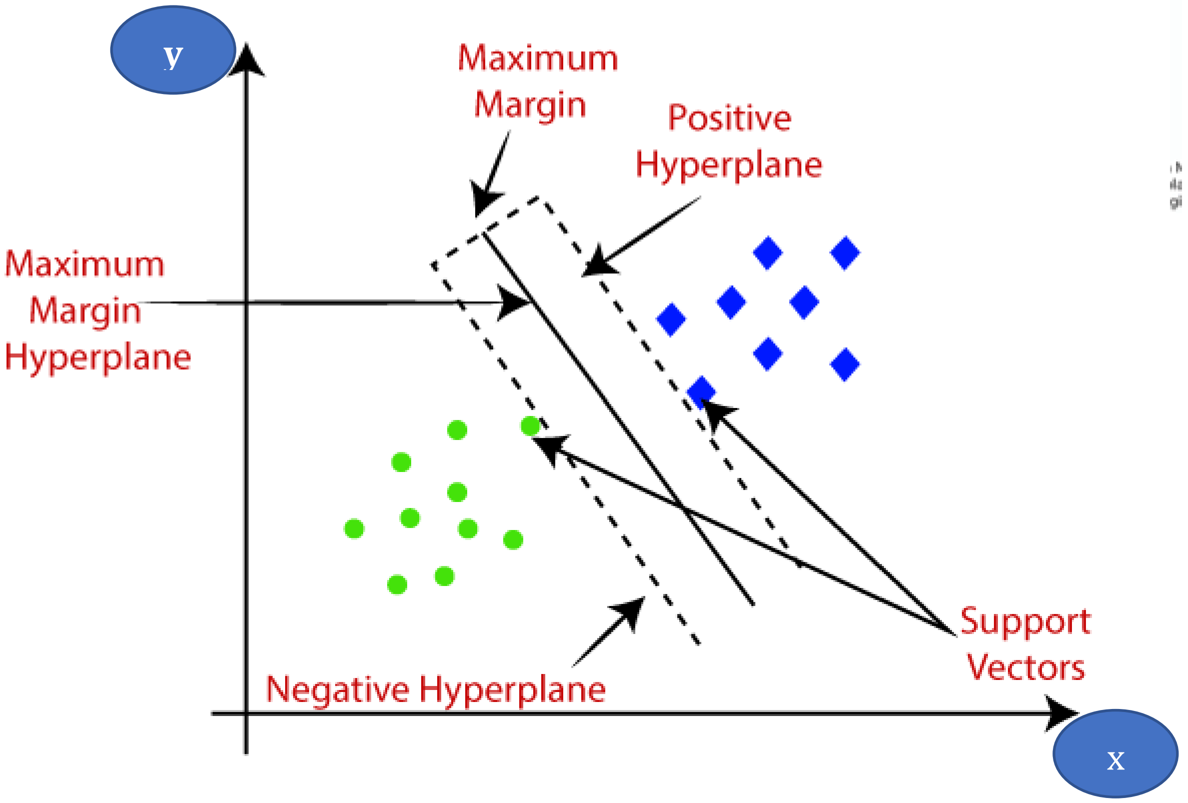


4. Support Vector Machine

- Supervised Machine Learning Model
- Used for both Classification and Regression
- Hyperplane
- Support Vectors

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression tasks. It is a powerful and versatile algorithm that aims to find an optimal hyperplane or decision boundary in a high-dimensional feature space to separate different classes or predict numerical values.

The fundamental idea behind SVM is to find the hyperplane that maximally separates the data points of different classes. The hyperplane is selected such that the distance between the hyperplane and the closest data points from each class, known as support vectors, is maximized. This distance is called the margin. The support vectors and the hyperplane are the key components of SVM. The support vectors are the crucial data points that influence the construction of the hyperplane, which in turn determines the separation between different classes and enables accurate classification or regression.



Support Vectors: In Support Vector Machine (SVM), support vectors are the data points that lie closest to the decision boundary, known as the hyperplane. These support vectors play a crucial role in defining the decision boundary and determining the optimal hyperplane that maximizes the margin.

The support vectors are the subset of training data points that have the most influence on the construction of the hyperplane. They are the points that are located on or near the margin, as well as the points that are misclassified. These data points are crucial because they define the separation between different classes and contribute to the calculation of the margin.

The choice of support vectors is determined during the training process of the SVM algorithm. The algorithm selects the support vectors based on their distance from the decision boundary. Only the support vectors are necessary to define the hyperplane and make predictions, rather than using all the training data points. This property of SVM makes it memory-efficient and computationally efficient.

Hyperplane: In SVM, the hyperplane is a decision boundary that separates different classes in the feature space. For binary classification tasks, the hyperplane is a (d-1)-dimensional subspace in a d-dimensional feature space.

In a linear SVM, the hyperplane is a linear combination of the input features. Mathematically, it can be represented as:

w^T x + b = 0

where w is the weight vector perpendicular to the hyperplane, x is the input feature vector, and b is the bias term. The weight vector w determines the orientation of the hyperplane, while the bias term b shifts the hyperplane.

The objective of SVM is to find the optimal hyperplane that maximizes the margin, which is the distance between the hyperplane and the nearest data points from each class, i.e., the support vectors. The hyperplane that achieves the maximum margin is considered the best decision boundary, as it provides better generalization to unseen data.

In cases where the data is not linearly separable, SVM uses the kernel trick to transform the feature space into a higher-dimensional space. In this higher-dimensional space, a hyperplane is sought to separate the transformed data. The kernel function computes the inner products of the transformed feature vectors without explicitly calculating the transformation. This allows SVM to capture complex non-linear decision boundaries.

For linearly separable data, SVM finds the hyperplane that achieves the maximum margin. However, when the data is not linearly separable, SVM uses a technique called the kernel trick to transform the original feature space into a higher-dimensional space, where the classes can be separated by a hyperplane.

The kernel trick allows SVM to implicitly map the data into a higher-dimensional space without explicitly calculating the transformed feature vectors. This is computationally efficient and enables SVM to capture complex non-linear relationships between features.

Advantages of Support Vector Machine:

- **Effective in high-dimensional spaces:** SVM performs well even in cases where the number of features is much greater than the number of samples. This makes it suitable for tasks involving a large number of features, such as text classification or image recognition.
- **Versatility:** SVM supports different kernel functions, such as linear, polynomial, and radial basis function (RBF), allowing flexibility in capturing non-linear relationships. This makes SVM adaptable to various types of data and problem domains.
- **Regularization:** SVM includes a regularization parameter (C) that controls the trade-off between maximizing the margin and minimizing the classification errors. This parameter helps prevent overfitting and allows the model to generalize well to unseen data.
- **Robust to outliers:** SVM is less sensitive to outliers compared to other classification algorithms like logistic regression. The use of support vectors, which are the closest data points to the decision boundary, makes the model less affected by outliers.

Limitations of Support Vector Machine:

- **Computationally intensive:** SVM can be computationally expensive, especially when dealing with large datasets. Training time and memory requirements can increase significantly as the number of samples and features grows.
- **Parameter selection:** SVM has several parameters, including the choice of kernel function, regularization parameter (C), and kernel-specific parameters. Selecting appropriate values for these parameters can be challenging and often requires careful tuning.
- **Interpretability:** While SVM can provide accurate predictions, it is not as interpretable as some other models like decision trees or logistic regression. The learned model does not directly provide insights into the relationship between individual features and the target variable.

Applications of Support Vector Machine:

- **Text and document classification:** SVM is widely used for tasks such as sentiment analysis, spam detection, topic classification, and document categorization in natural language processing.
- **Image classification:** SVM has been successfully applied to image recognition tasks, including object detection, facial expression recognition, and handwritten digit recognition.
- **Bioinformatics:** SVM is used in protein structure prediction, gene expression analysis, and disease classification based on genomic data.
- **Financial analysis:** SVM can be applied to credit scoring, stock market prediction, fraud detection, and anomaly detection in financial data.
- **Medical diagnosis:** SVM has been employed in medical diagnosis, including cancer classification, disease prognosis, and identification of genetic markers.
- **Remote sensing:** SVM is used in satellite image analysis, land cover classification, and pattern recognition in remote sensing applications.

Implementing SVM using Python

Dataset Required

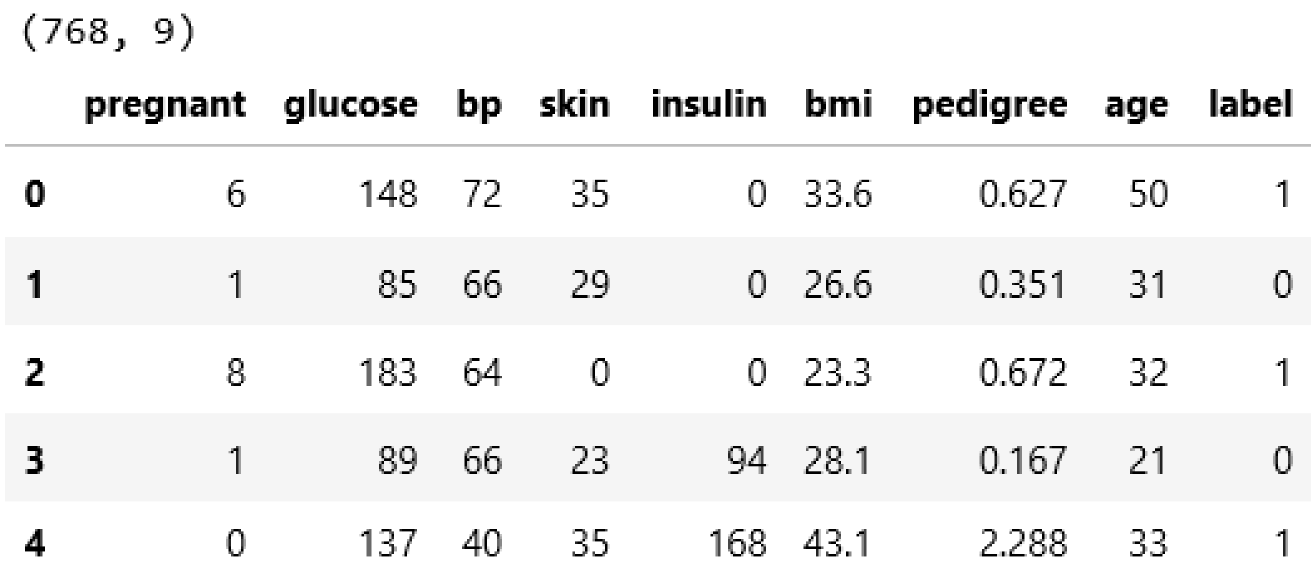
https://drive.google.com/file/d/1V6yFU3nDdx9R56yOzy6GxHPq-Dav2A4K/view?usp=share_link

Importing Required Libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn import metrics
import seaborn as sn
```

Importing (Reading) Datasets

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin',
             'bmi', 'pedigree', 'age', 'label']
data = pd.read_csv('/content/diabets.csv', header= None,
       names=col_names)
print(data.shape)
data.head()
```

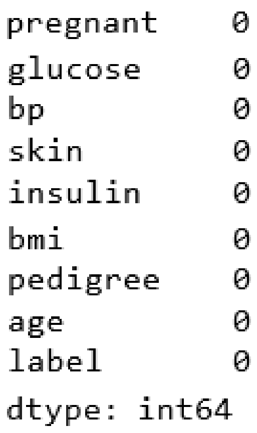


```
(768, 9)
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Checking for any null values in dataset

```
data.isnull().sum()
```



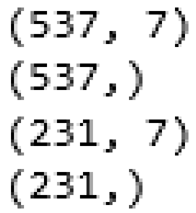
```
pregnant    0
glucose     0
bp          0
skin        0
insulin     0
bmi         0
pedigree    0
age         0
label       0
dtype: int64
```

Assigning dependent and independent variables

```
feature_cols = ['pregnant','insulin', 'bmi',
                'age','glucose','bp', 'pedigree']
x=data[feature_cols]
y=data.label
```

splitting the dataset into Training and Testing Dataset

```
x_train, x_test, y_train, y_test = train_test_split(x,y,
     test_size=0.3, random_state=5)
display(x_train.shape, y_train.shape, x_test.shape,
     y_test.shape)
```



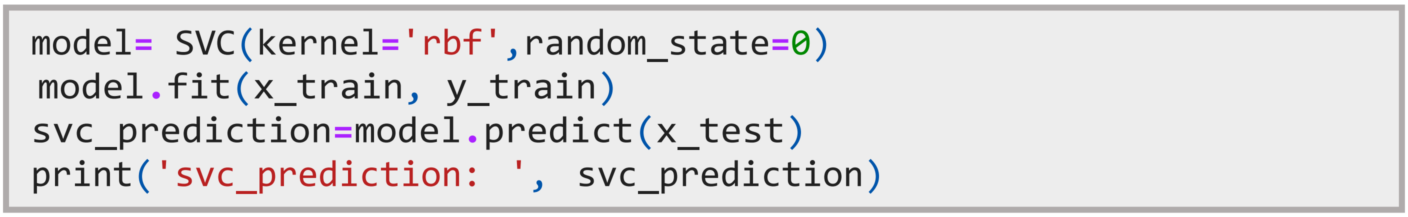
```
(537, 7)
(537,)
(231, 7)
(231,)
```

Preprocessing Data with StandardScaler

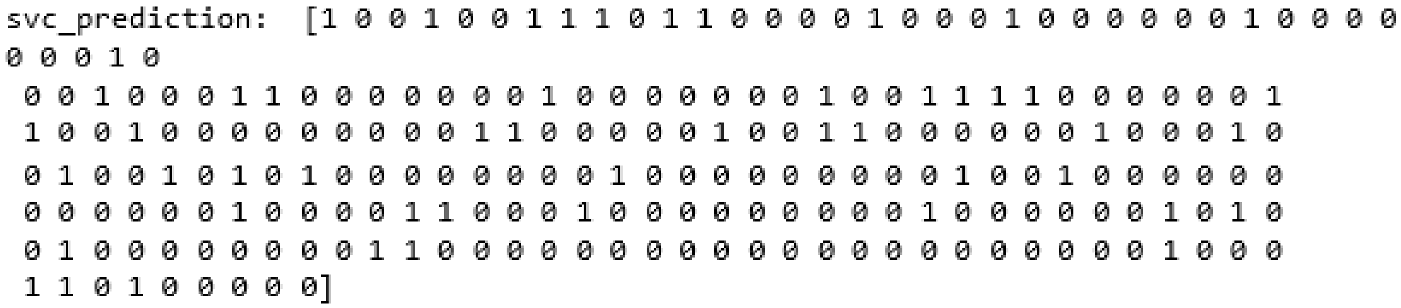
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train=sc.fit_transform(x_train)
x_test=sc.transform(x_test)
```

Standardization (Z-score normalization) scales the features of a dataset so that they have zero mean and unit variance. This transformation centers the data around the mean and scales it by the standard deviation. It does not enforce a specific range for the transformed values. Normalization, on the other hand, scales the features to a specific range, often between 0 and 1 or -1 and 1. It is achieved by dividing each value by the maximum value in the feature range or by applying other normalization techniques.

Fitting the Model (SVM) using 'rbf' kernel

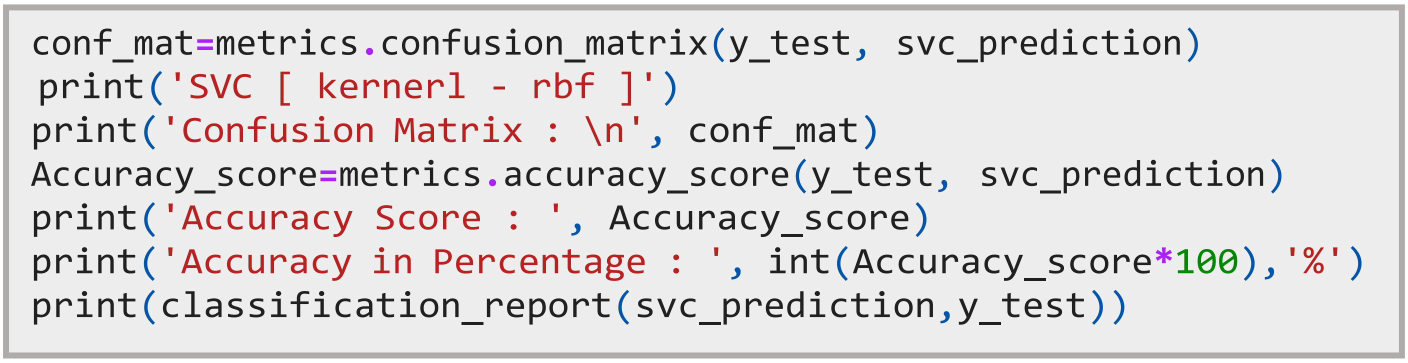


```
model= SVC(kernel='rbf',random_state=0)
 model.fit(x_train, y_train)
svc_prediction=model.predict(x_test)
print('svc_prediction:  ', svc_prediction)
```

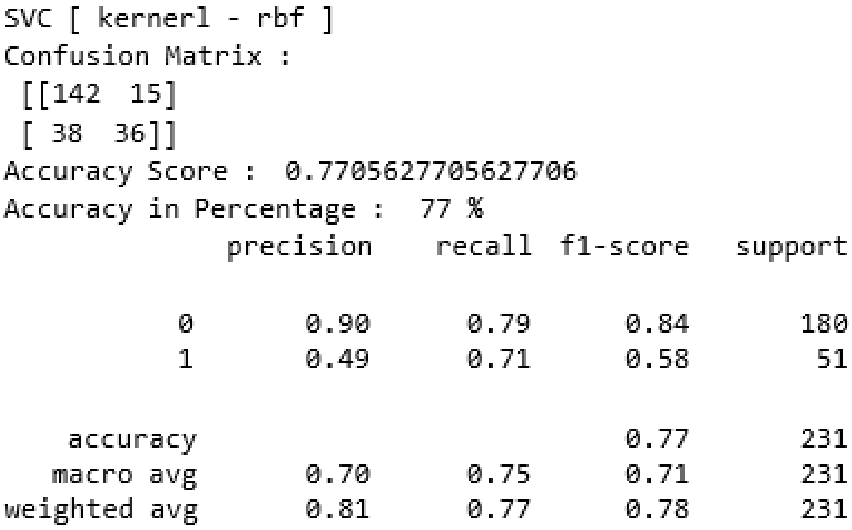


```
svc_prediction:  [1 0 0 1 0 0 1 1 1 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0
 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 1
 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0
 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0
 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
 1 1 0 1 0 0 0 0 0]
```

Evaluation Metrics for 'rbf' kernel

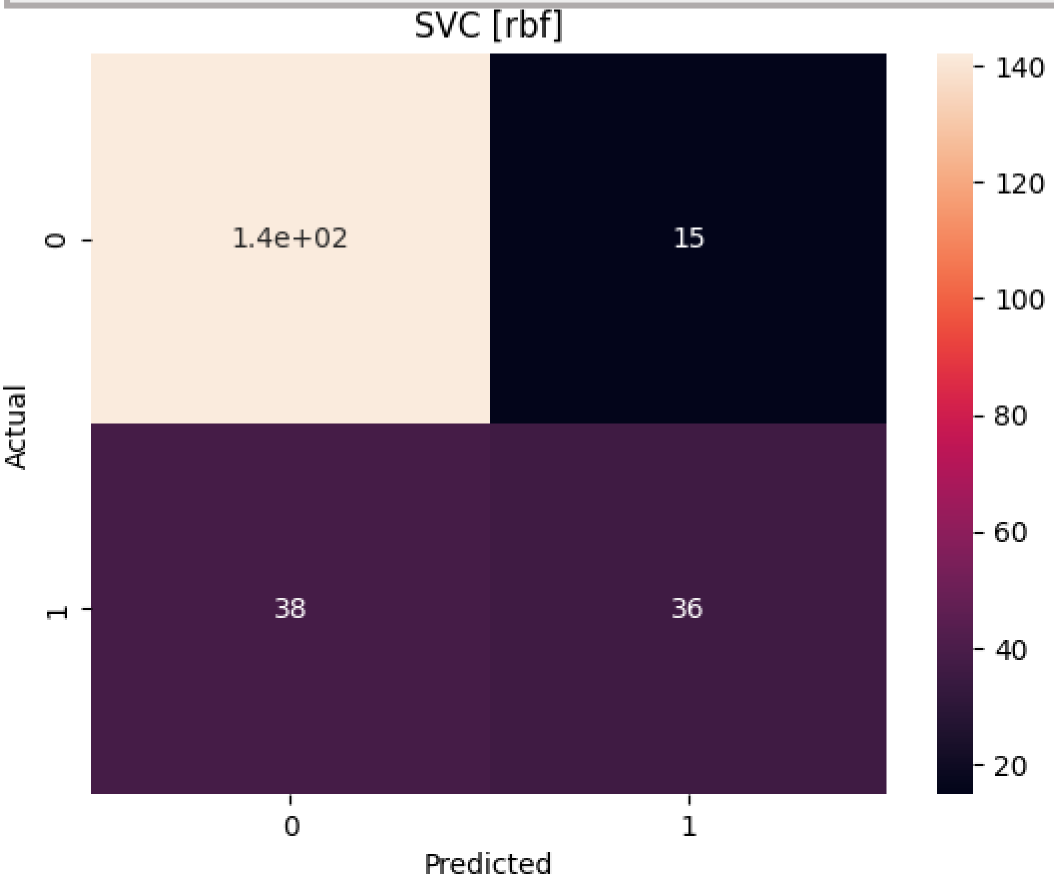


```
conf_mat=metrics.confusion_matrix(y_test, svc_prediction)
 print('SVC [ kernerl - rbf ]')
print('Confusion Matrix : \n', conf_mat)
Accuracy_score=metrics.accuracy_score(y_test, svc_prediction)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ', int(Accuracy_score*100),'%')
print(classification_report(svc_prediction,y_test))
```



```
SVC [ kernerl - rbf ]
Confusion Matrix :
 [[142  15]
 [ 38  36]]
Accuracy Score :  0.7705627705627706
Accuracy in Percentage :   77 %
              precision    recall  f1-score   support

           0       0.90      0.79      0.84       180
           1       0.49      0.71      0.58        51

    accuracy                           0.77       231
   macro avg       0.70      0.75      0.71       231
weighted avg       0.81      0.77      0.78       231
```

```
conf_mat=pd.crosstab(y_test, y_pred, rownames=['Actual'],
    colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True).set(title='SVC [rbf]')
```

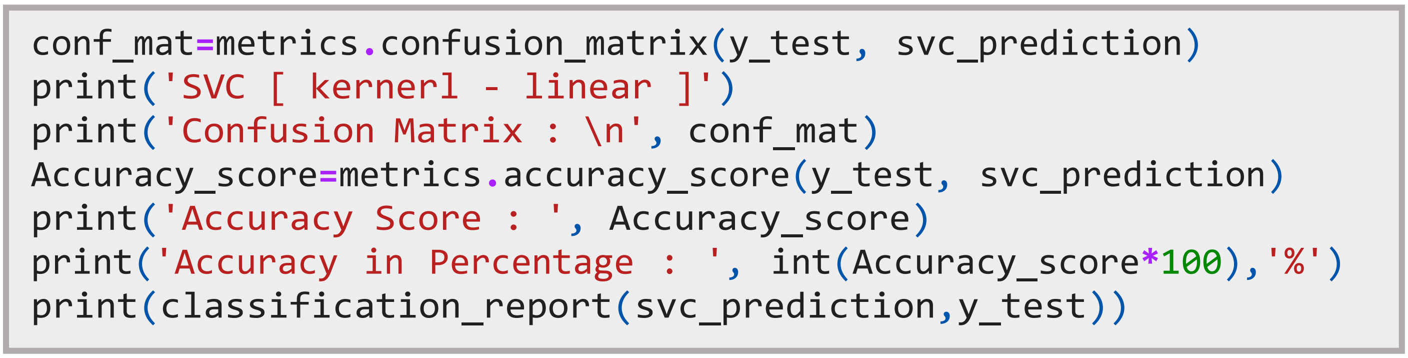


Fitting the Model (SVM) using 'Linear' kernel

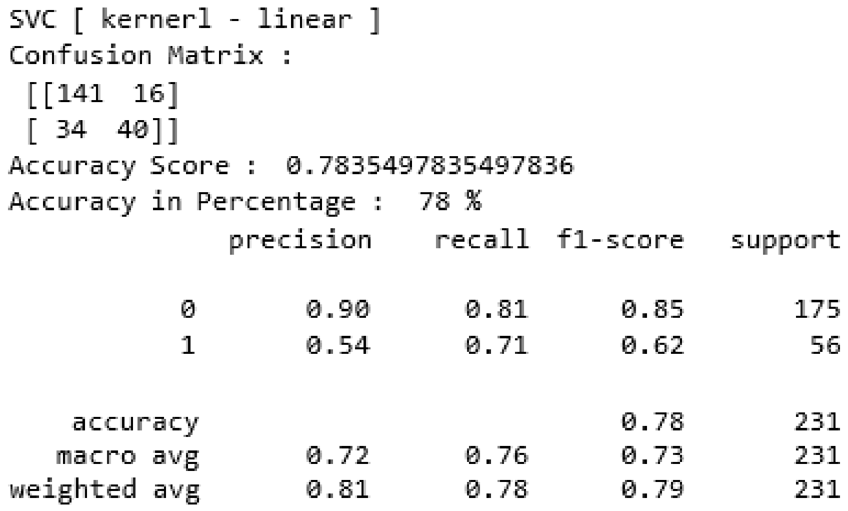
```
model= SVC(kernel='linear',random_state=0)
model.fit(x_train, y_train)
svc_prediction=model.predict(x_test)
print('svc_prediction: ', svc_prediction)
```

```
svc_prediction:  [1 0 0 1 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0
 0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 1
 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0
 0 1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0
 0 0 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0
 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
 0 1 1 1 0 0 0 0 0]
```

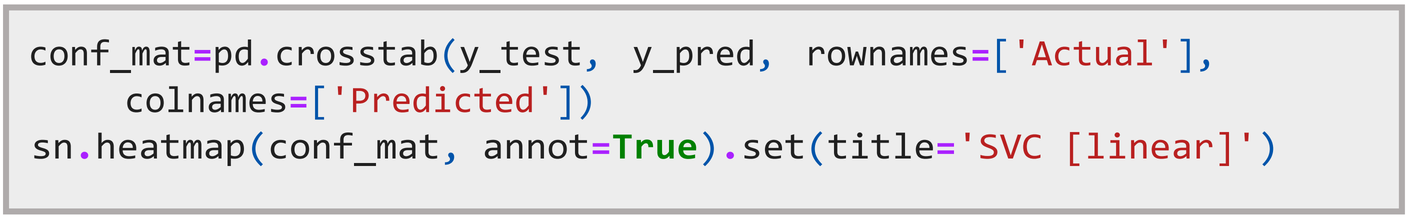
Evaluation Metrics for 'Linear' kernel



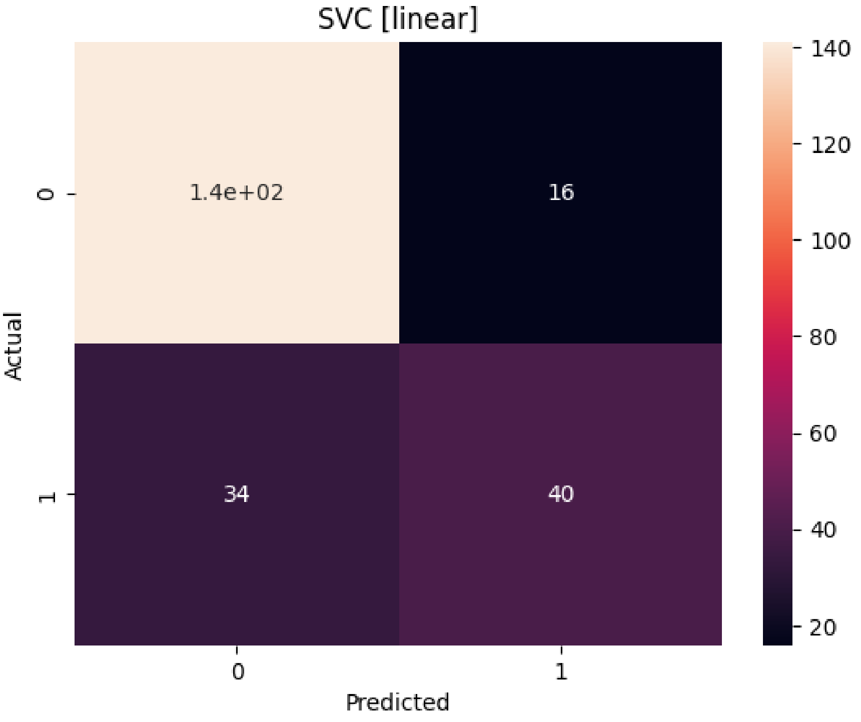
```
conf_mat=metrics.confusion_matrix(y_test, svc_prediction)
print('SVC [ kernerl - linear ]')
print('Confusion Matrix : \n', conf_mat)
Accuracy_score=metrics.accuracy_score(y_test, svc_prediction)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ', int(Accuracy_score*100),'%')
print(classification_report(svc_prediction,y_test))
```



```
SVC [ kernerl - linear ]
Confusion Matrix :
 [[141  16]
 [ 34  40]]
Accuracy Score :  0.7835497835497836
Accuracy in Percentage :  78 %
              precision    recall  f1-score   support

           0       0.90      0.81      0.85       175
           1       0.54      0.71      0.62        56

    accuracy                           0.78       231
   macro avg       0.72      0.76      0.73       231
weighted avg       0.81      0.78      0.79       231
```



```
conf_mat=pd.crosstab(y_test, y_pred, rownames=['Actual'],
     colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True).set(title='SVC [linear]')
```

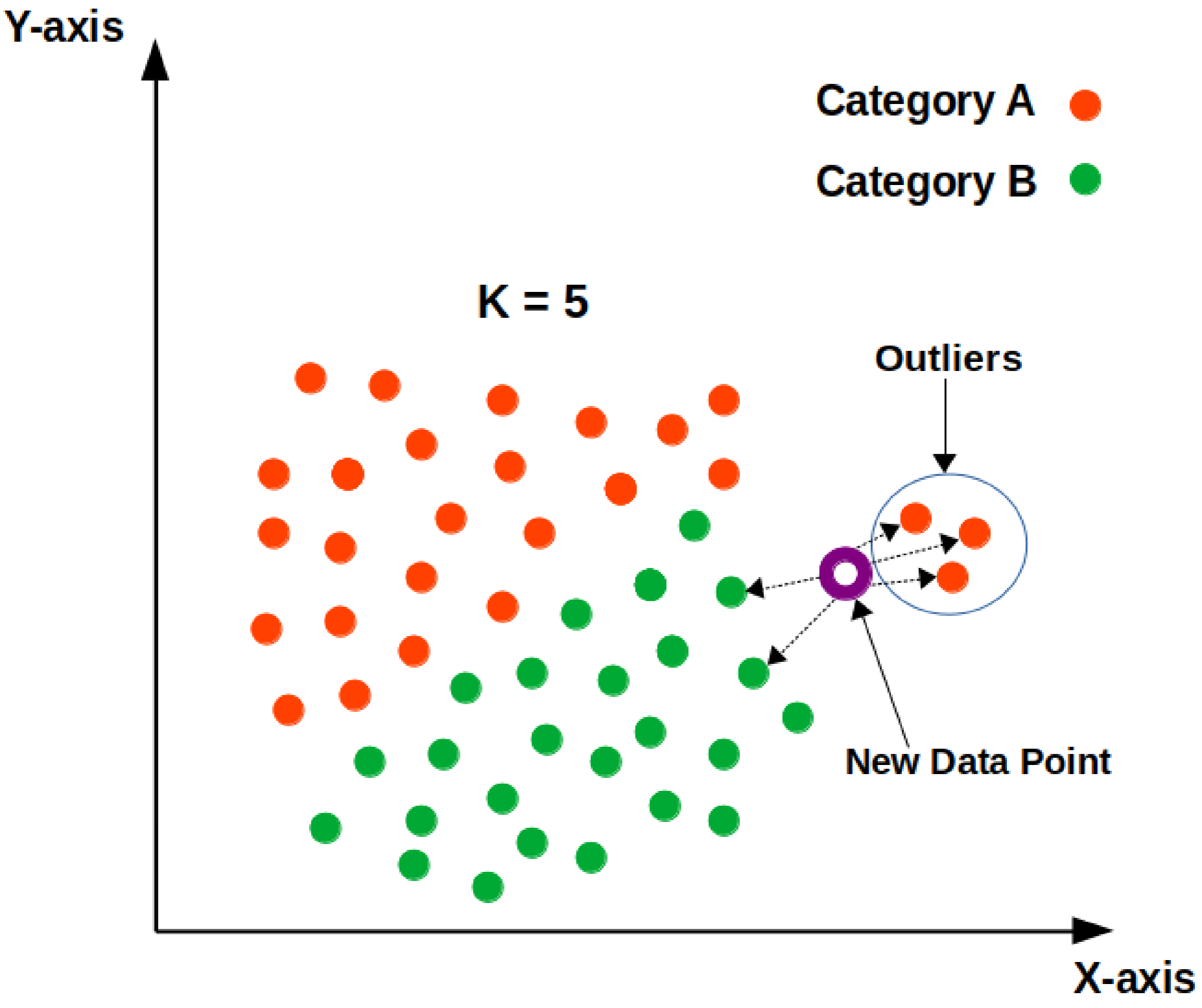


5. K Nearest Neighbors

- Used for both classification and regression
- Distance based model

K-nearest neighbors (KNN) is a non-parametric machine learning algorithm used for both classification and regression tasks. It is a simple yet powerful algorithm that makes predictions based on the similarity of input data to its neighboring data points.

Theory: The KNN algorithm works based on the principle that similar data points tend to share the same class or have similar output values. The algorithm stores the entire training dataset in memory and uses it during the prediction phase. When a new data point is provided, the algorithm calculates the distance between that point and all other data points in the training set. The distance metric used is typically Euclidean distance, although other distance metrics can be employed.



KNN algorithm:

1. Determine the number of neighbors (K) to consider, usually specified by the user.
2. Calculate the distance between the new data point and all other data points in the training set.
3. Select the K data points with the shortest distances (i.e., the K nearest neighbors).
4. For classification tasks, assign the class label that is most frequent among the K nearest neighbors to the new data point. For regression tasks, compute the average or weighted average of the output values of the K nearest neighbors.
5. Output the predicted class label or regression value for the new data point.

Advantages of KNN:

- **Simplicity:** KNN is easy to understand and implement. It does not require any assumptions about the underlying data distribution or model structure.
- **Versatility:** KNN can be applied to both classification and regression problems. It can handle both numerical and categorical data.
- **Adaptability:** KNN is a lazy learner, meaning it does not perform a training phase. This makes it suitable for dynamic or changing environments, as the model can be updated with new data points easily.
- **Non-linearity:** KNN can capture complex, non-linear relationships between the input features and the target variable.

Limitations of KNN:

- **Computational Complexity:** As the algorithm compares the new data point with all training data points, the computational cost can be high, especially for large datasets.
- **Sensitivity to Feature Scaling:** KNN calculates distances based on feature values. If the features have different scales, features with larger values can dominate the distance calculation, leading to biased results. It is essential to normalize or standardize the features before applying KNN.

- **Determining the Optimal K:** Choosing the optimal number of neighbors (K) is subjective and depends on the dataset and problem at hand. A small K may result in overfitting, while a large K may introduce more noise and dilute the local patterns.

Applications of KNN:

- **Recommender Systems:** KNN can be used to build recommendation engines by finding similar users or items based on their features or preferences.
- **Image Recognition:** KNN can be applied to image classification tasks by comparing the pixel values or feature vectors of images to identify similar objects or patterns.
- **Anomaly Detection:** KNN can detect outliers or anomalies in data by identifying data points that are significantly different from their neighbors.
- **Text Classification:** KNN can classify text documents based on their word frequencies or vector representations by measuring the similarity between documents.
- **Healthcare:** KNN can assist in medical diagnosis by finding similar patient cases or medical images for comparison and decision support.

Implementing KNN using Python

Dataset Required

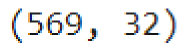
https://drive.google.com/file/d/1865t5MZPQn53A5bSYMN86D0BmLSBxan-/view?usp=share_link

Importing Required Libraries

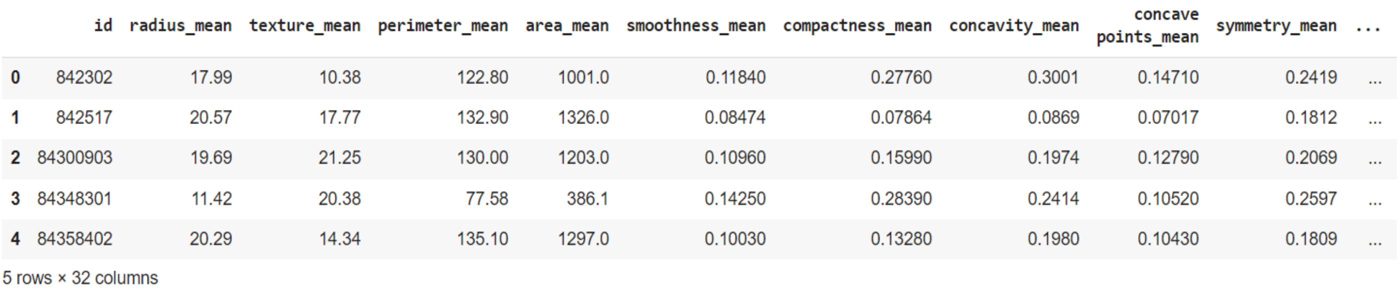
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn import metrics
import seaborn as sn
```

Importing (Reading) Datasets

```
data = pd.read_csv('/content/Breast Cancer Detection
Classification Master.csv')
print(data.shape)
data.head()
```



```
(569, 32)
```

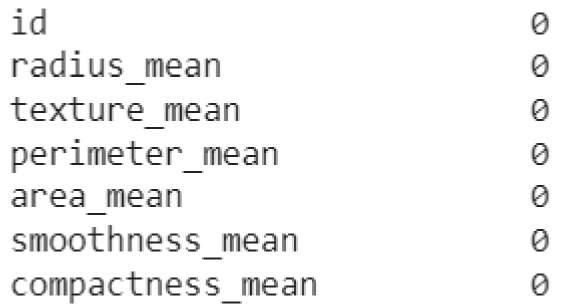


	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...
0	842302	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...
1	842517	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...
2	84300903	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...
3	84348301	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...
4	84358402	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...

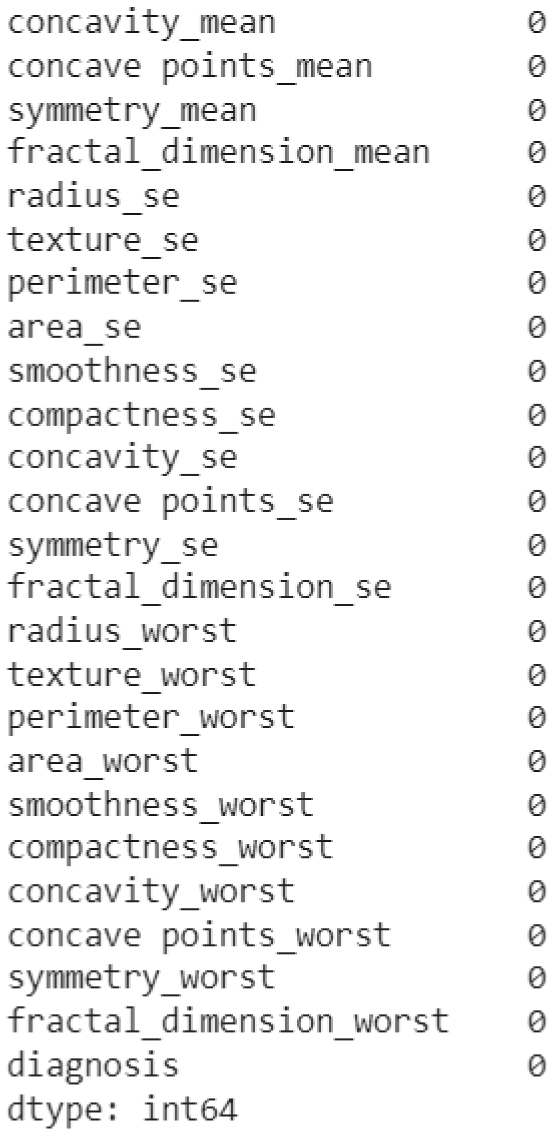
5 rows × 32 columns

Checking for any null values in dataset

```
data.isnull().sum()
```



```
id                         0
radius_mean                0
texture_mean               0
perimeter_mean             0
area_mean                  0
smoothness_mean            0
compactness_mean           0
```



```
concavity_mean            0
concave points_mean       0
symmetry_mean             0
fractal_dimension_mean    0
radius_se                 0
texture_se                0
perimeter_se              0
area_se                   0
smoothness_se             0
compactness_se            0
concavity_se              0
concave points_se         0
symmetry_se               0
fractal_dimension_se      0
radius_worst              0
texture_worst             0
perimeter_worst           0
area_worst                0
smoothness_worst          0
compactness_worst         0
concavity_worst           0
concave points_worst      0
symmetry_worst            0
fractal_dimension_worst   0
diagnosis                 0
dtype: int64
```

Assigning dependent and independent variables

```
x=data.iloc[:, :-1].values
y=data.iloc[:, -1].values
```

splitting the dataset into Training and Testing Dataset

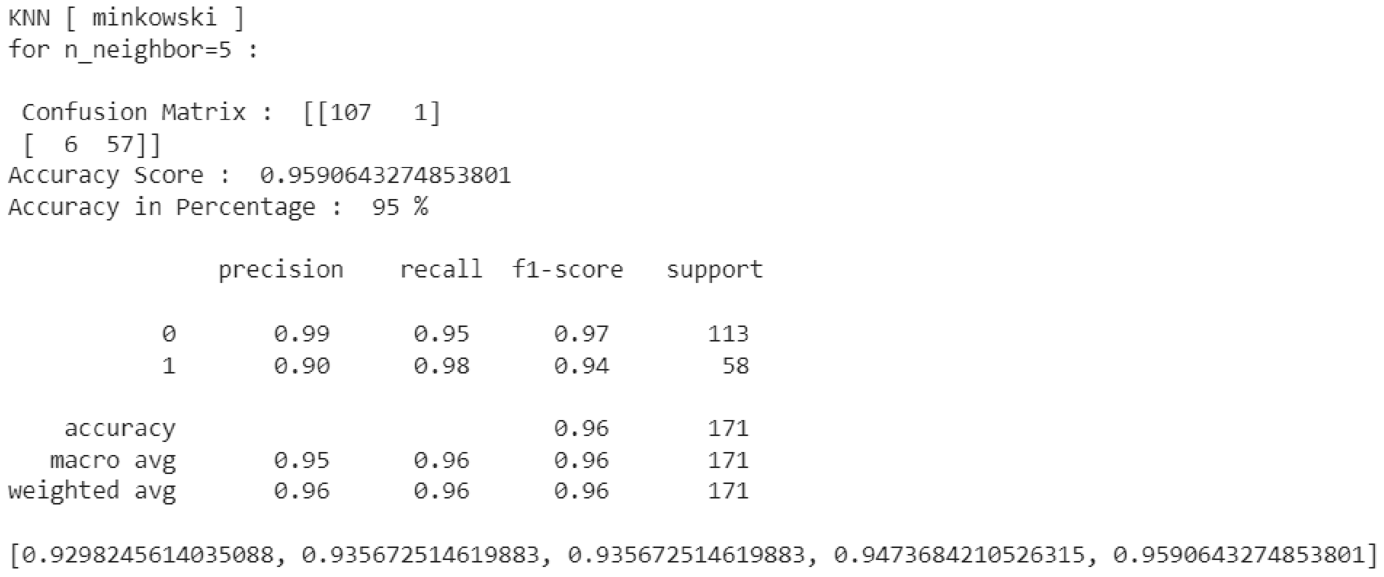
```
x_train, x_test, y_train, y_test = train_test_split(x,y,
test_size=0.3, random_state=0)
# display(x_train.shape, y_train.shape, x_test.shape,
y_test.shape)
```

Preprocessing Data with StandardScaler

```
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
x_train=sc.fit_transform(x_train)
x_test=sc.transform(x_test)
```

Training the Model using KNN (minkoski):

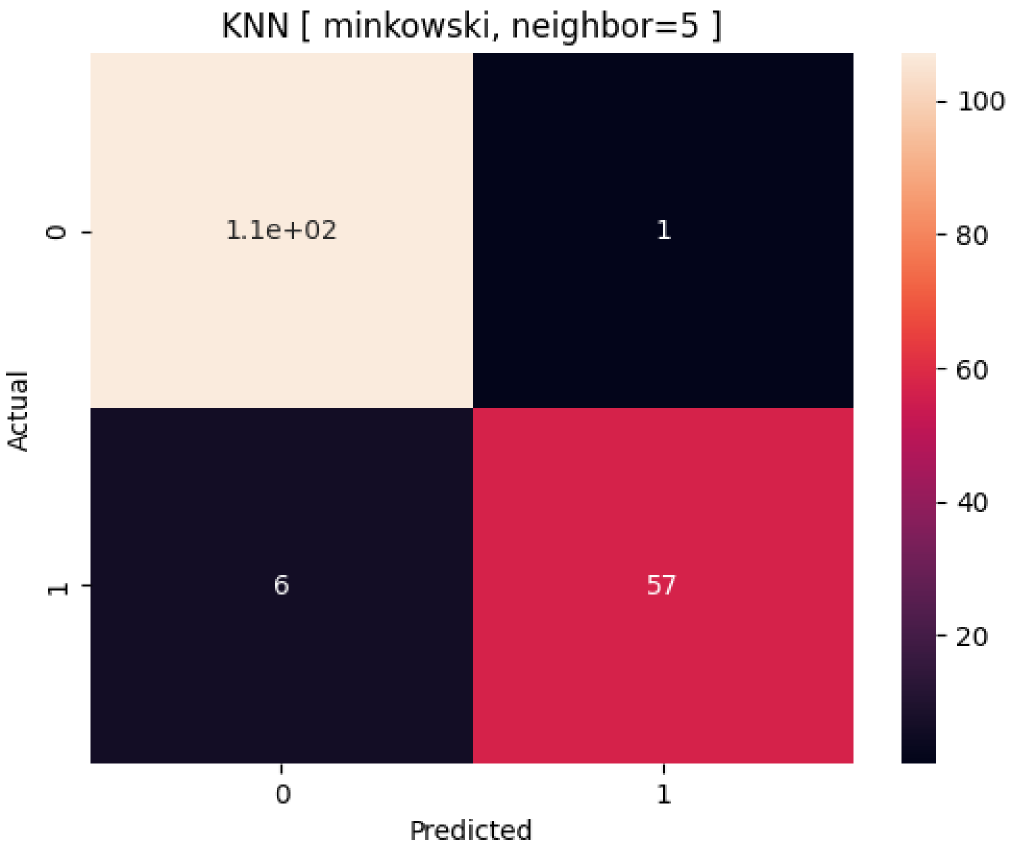
```
results=[]
for i in [1,2,3,4,5]:
    model = KNeighborsClassifier(n_neighbors=i,
metric='minkowski', p=2)
    model.fit(x_train, y_train)
    y_pred=model.predict(x_test)
    Accuracy_score=metrics.accuracy_score(y_test, y_pred)
    results.append(Accuracy_score)

print('KNN [ minkowski ]')
print('for n_neighbor=5 : ')
conf_mat=metrics.confusion_matrix(y_test, y_pred)
print('\n Confusion Matrix : ', conf_mat)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ',
int(Accuracy_score*100),'%')
print('\n',classification_report(y_pred,y_test))

print(results)
```



```
KNN [ minkowski ]
for n_neighbor=5 :

 Confusion Matrix :  [[107   1]
 [  6  57]]
Accuracy Score :  0.9590643274853801
Accuracy in Percentage :  95 %

               precision    recall  f1-score   support

           0       0.99      0.95      0.97       113
           1       0.90      0.98      0.94        58

    accuracy                           0.96       171
   macro avg       0.95      0.96      0.96       171
weighted avg       0.96      0.96      0.96       171

[0.9298245614035088, 0.935672514619883, 0.935672514619883, 0.9473684210526315, 0.9590643274853801]
```

Evaluation Metrics

```
conf_mat=pd.crosstab(y_test, y_pred, rownames=['Actual'],
colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True).set(title='KNN [ minkowski,
neighbor=5 ]')
```



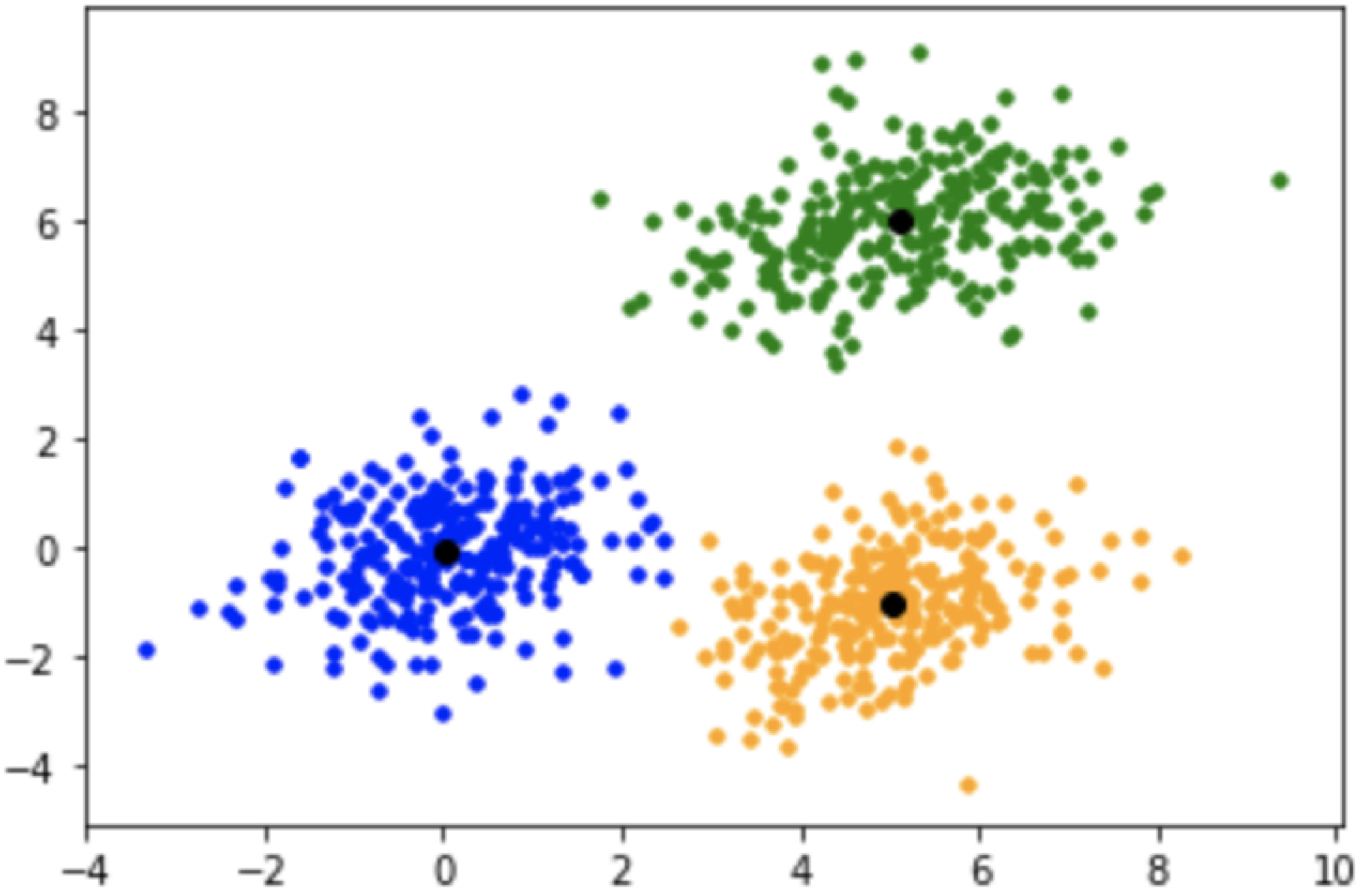
```
models = pd.DataFrame({
    'n_neighbors': ['1', '2','3','4','5'],
    'Accuracy Score':
[results[0],results[1],results[2],results[3],results[4]]})
models.sort_values(by='Accuracy Score')
print(models.to_string(index=False))
```

```
n_neighbors  Accuracy Score
          1        0.929825
          2        0.935673
          3        0.935673
          4        0.947368
          5        0.959064
```

6. K Means Clustering

- Unsupervised Learning
- Used for clustering, not for classification or regression
- K - Number
- Clusters

K-means clustering is an unsupervised machine learning algorithm used to partition a dataset into K distinct clusters. The goal is to group similar data points together and ensure that data points within the same cluster are more similar to each other than to those in other clusters. The algorithm accomplishes this by iteratively assigning data points to the nearest cluster centroid and updating the centroids based on the assigned points.



K-means clustering algorithm:

1. **Initialization:** Select the number of clusters, K, and randomly initialize K cluster centroids in the feature space.

2. **Assignment:** Assign each data point to the nearest centroid based on a distance metric, typically Euclidean distance. Each data point belongs to the cluster with the closest centroid.
3. **Update:** Recalculate the centroids of each cluster by taking the mean of all the data points assigned to that cluster.
4. **Repeat steps 2 and 3 until convergence:** Iterate steps 2 and 3 until the cluster assignments no longer change significantly or a maximum number of iterations is reached.
5. **Finalization:** Once the algorithm converges, the final centroids represent the cluster centers, and each data point is assigned to a specific cluster.

Advantages of K-means clustering:

- **Simplicity:** K-means is a relatively simple and intuitive algorithm to understand and implement. It is computationally efficient and can handle large datasets efficiently.
- **Scalability:** K-means clustering is scalable to large datasets as it has a linear computational complexity. It can be applied to a wide range of data sizes and dimensions.
- **Versatility:** K-means can be applied to various types of data and is not restricted to any specific data distribution. It is effective in finding clusters of different shapes and sizes.
- **Interpretable results:** The cluster centroids obtained from K-means are interpretable and can provide insights into the structure and patterns present in the data.

Limitations of K-means clustering:

- **Dependency on initial centroid positions:** K-means is sensitive to the initial placement of centroids. Different initializations can result in different cluster assignments and outcomes. To mitigate this, multiple runs with different initializations are often performed, and the best clustering solution is chosen.
- **Fixed number of clusters:** K-means requires the user to specify the number of clusters, K, in advance. If the true number of clusters is unknown, determining the appropriate value of K can be challenging. Incorrectly specified K may lead to suboptimal clustering results.

- **Sensitive to outliers:** K-means is sensitive to outliers as they can significantly impact the centroid calculation. Outliers may be assigned to inappropriate clusters or form their own clusters, affecting the overall clustering quality.
- **Limited to linear boundaries:** K-means assumes that clusters are isotropic, spherical, and have equal variance. It struggles to handle non-linear cluster boundaries and clusters of different shapes and sizes. Other clustering algorithms like DBSCAN or hierarchical clustering may be more appropriate for such scenarios.

Applications of K-means clustering:

- **Customer segmentation:** K-means clustering is widely used in market research to segment customers based on their behavior, preferences, or demographics. This helps businesses target specific customer groups with tailored marketing strategies.
- **Image compression:** K-means can be used to compress images by reducing the number of colors required to represent an image. By clustering similar colors together and using fewer colors to represent each cluster, image size can be reduced without significant loss of quality.
- **Anomaly detection:** K-means clustering can be employed for detecting anomalies or outliers in datasets. By considering data points that do not fit well into any cluster or are assigned to small clusters, it is possible to identify unusual or potentially suspicious instances. This is useful in fraud detection, network intrusion detection, and outlier analysis.
- **Recommendation Systems:** K-means clustering can contribute to building recommendation systems. By clustering users or items based on their preferences or behavior, it becomes possible to make personalized recommendations to users or group similar items together.
- **Market Segmentation:** K-means clustering is utilized in market research to segment markets or consumer populations. By clustering individuals or regions based on socioeconomic factors, purchasing behavior, or lifestyle, businesses can target specific segments with tailored marketing strategies.

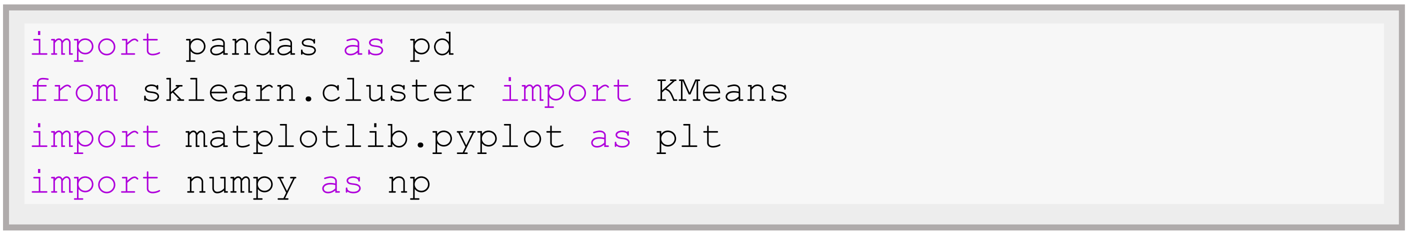
- **Social Network Analysis:** K-means clustering can be used in social network analysis to identify communities or groups of individuals with similar characteristics or interaction patterns. It helps in understanding the structure and dynamics of social networks and can be applied in various domains such as marketing, sociology, and online social platforms.

Implementing K-Means Clustering using Python

Dataset Required

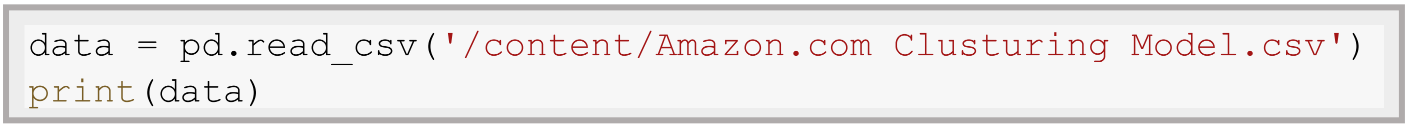
https://drive.google.com/file/d/1s-aZWEqNCPTBWCK6qACmbqH6lMDt6Hyp/view?usp=sharing

Importing Required Libraries

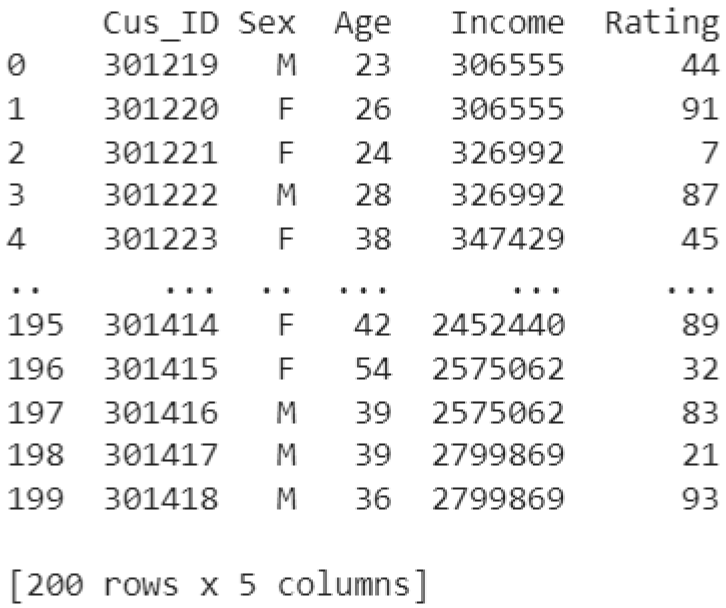


```
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
```

Importing (Reading) Datasets

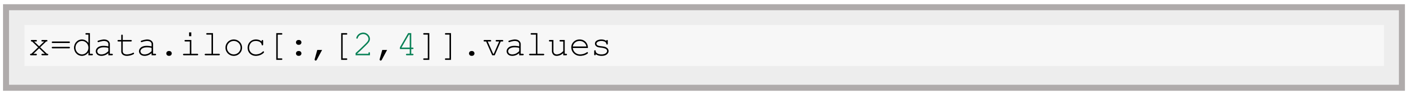


```
data = pd.read_csv('/content/Amazon.com Clusturing Model.csv')
print(data)
```



```
     Cus_ID Sex  Age   Income  Rating
0    301219   M   23   306555      44
1    301220   F   26   306555      91
2    301221   F   24   326992       7
3    301222   M   28   326992      87
4    301223   F   38   347429      45
..      ...  ..  ...      ...     ...
195  301414   F   42  2452440      89
196  301415   F   54  2575062      32
197  301416   M   39  2575062      83
198  301417   M   39  2799869      21
199  301418   M   36  2799869      93

[200 rows x 5 columns]
```

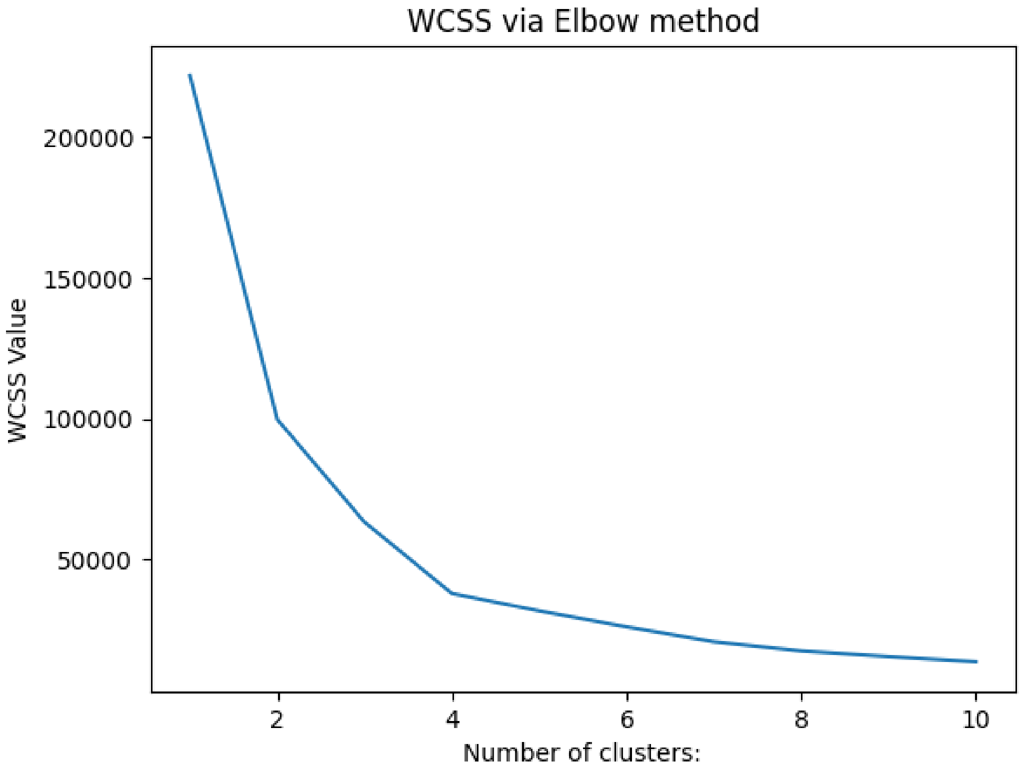
Assigning independent variable



```
x=data.iloc[:,[2,4]].values
```

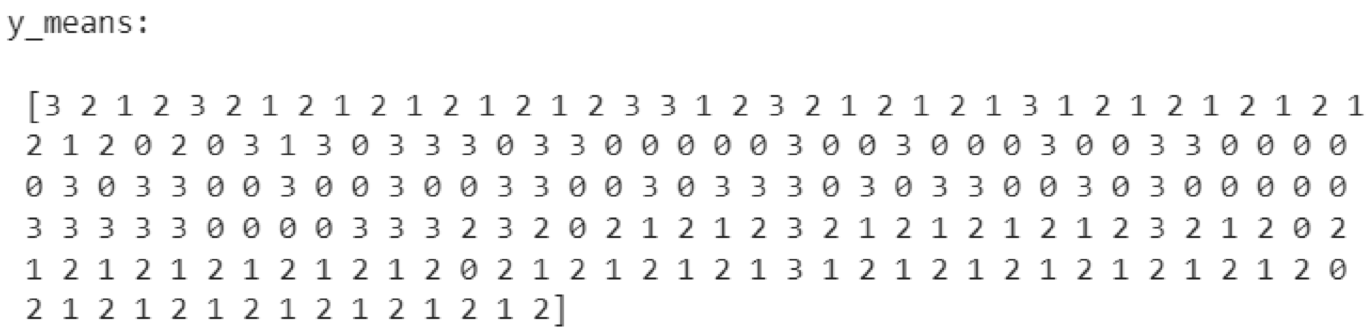
Number of Clusters via Elbow Method

```
wcss=[]
for i in range(1,11):
  model=KMeans(n_clusters=i,init='k-means++',random_state=21)
  model.fit(x)
  wcss.append(model.inertia_)
plt.plot(range(1,11),wcss)
plt.title('WCSS via Elbow method')
plt.xlabel('Number of clusters:')
plt.ylabel('WCSS Value')
plt.show()
```



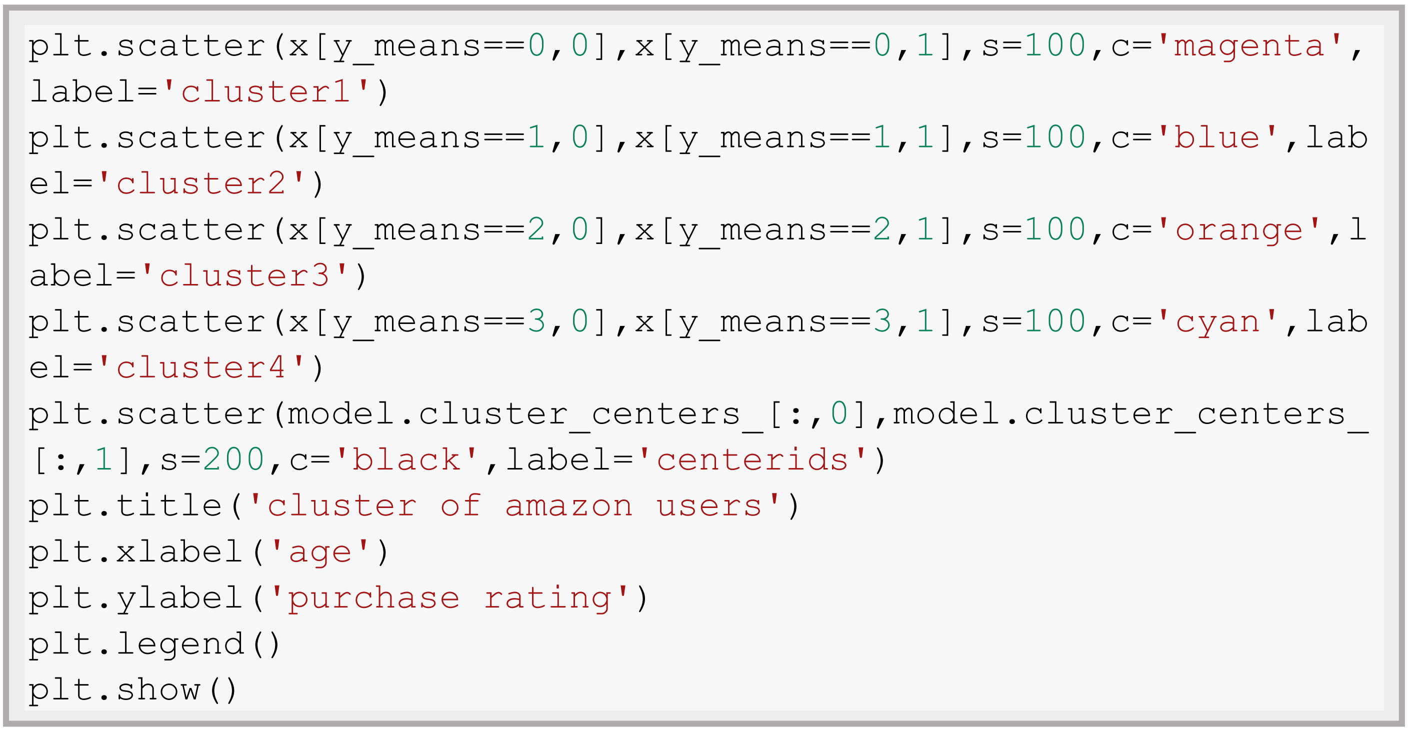
K-Means Clustering Training on Training set

```
model = KMeans(n_clusters=4,init='k-means++',random_state=42)
y_means=model.fit_predict(x)
print("y_means:\n\n",y_means)
```

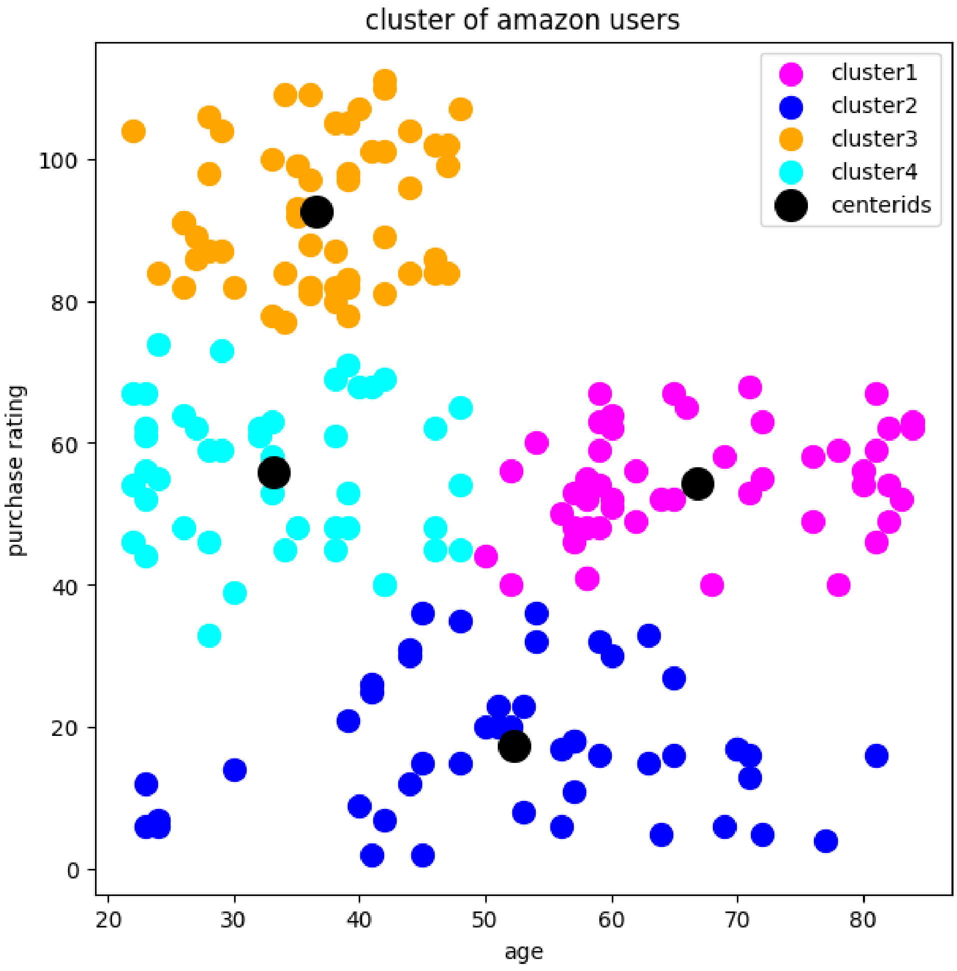


```
y_means:

 [3 2 1 2 3 2 1 2 1 2 1 2 1 2 1 2 3 3 1 2 3 2 1 2 1 2 1 3 1 2 1 2 1 2 1 2 1
 2 1 2 0 2 0 3 1 3 0 3 3 3 0 3 3 0 0 0 0 0 3 0 0 3 0 0 0 3 0 0 3 3 0 0 0 0
 0 3 0 3 3 0 0 3 0 0 3 0 0 3 3 0 0 3 0 3 3 3 0 3 0 3 3 0 0 3 0 3 0 0 0 0 0
 3 3 3 3 3 0 0 0 0 3 3 3 2 3 2 0 2 1 2 1 2 3 2 1 2 1 2 1 2 1 2 3 2 1 2 0 2
 1 2 1 2 1 2 1 2 1 2 1 2 0 2 1 2 1 2 1 2 1 3 1 2 1 2 1 2 1 2 1 2 1 2 1 2 0
 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2]
```

Scattering the Clusters



```
plt.scatter(x[y_means==0,0],x[y_means==0,1],s=100,c='magenta',label='cluster1')
plt.scatter(x[y_means==1,0],x[y_means==1,1],s=100,c='blue',label='cluster2')
plt.scatter(x[y_means==2,0],x[y_means==2,1],s=100,c='orange',label='cluster3')
plt.scatter(x[y_means==3,0],x[y_means==3,1],s=100,c='cyan',label='cluster4')
plt.scatter(model.cluster_centers_[:,0],model.cluster_centers_[:,1],s=200,c='black',label='centerids')
plt.title('cluster of amazon users')
plt.xlabel('age')
plt.ylabel('purchase rating')
plt.legend()
plt.show()
```



7. Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction technique commonly used in machine learning and data analysis. It aims to transform a dataset containing a large number of correlated variables into a smaller set of uncorrelated variables, known as principal components. PCA achieves this by identifying the directions, or principal axes, along which the data varies the most.

Steps involved in PCA:

1. **Data Standardization:** The first step in PCA is to standardize the data. This involves scaling the features so that they have zero mean and unit variance. Standardization is necessary to ensure that variables with larger scales do not dominate the analysis.

2. **Covariance Matrix Calculation:** Once the data is standardized, the next step is to calculate the covariance matrix. The covariance matrix represents the relationships between variables and measures how they vary together. It is a square matrix where each element represents the covariance between two variables.

3. **Eigenvector and Eigenvalue Calculation:** After computing the covariance matrix, the next step is to calculate the eigenvectors and eigenvalues of the matrix. Eigenvectors represent the directions or axes of the data, while eigenvalues represent the amount of variance explained by each eigenvector. The eigenvectors and eigenvalues are calculated using linear algebra techniques.

4. **Selection of Principal Components:** The eigenvectors are sorted in descending order based on their corresponding eigenvalues. The eigenvector with the highest eigenvalue represents the direction of maximum variance in the data and is considered the first principal component (PC1). The second principal component (PC2) is the eigenvector with the second-highest eigenvalue, and so on. The number of principal components to retain is a decision made by the analyst, typically based on the amount of variance they want to explain.

5. **Projection of Data onto Principal Components:** In this step, the original data is projected onto the selected principal components. Each data point is represented by a vector of values

along the principal components. This projection allows us to reduce the dimensionality of the dataset, as we can discard the principal components with lower importance.

6. **Reconstruction of Data:** If required, the projected data can be reconstructed back into the original feature space. This involves multiplying the projected data by the transposed eigenvectors and adding back the mean values that were subtracted during standardization. The reconstructed data will have reduced dimensions but will closely resemble the original data.

Applications of PCA in Machine Learning:

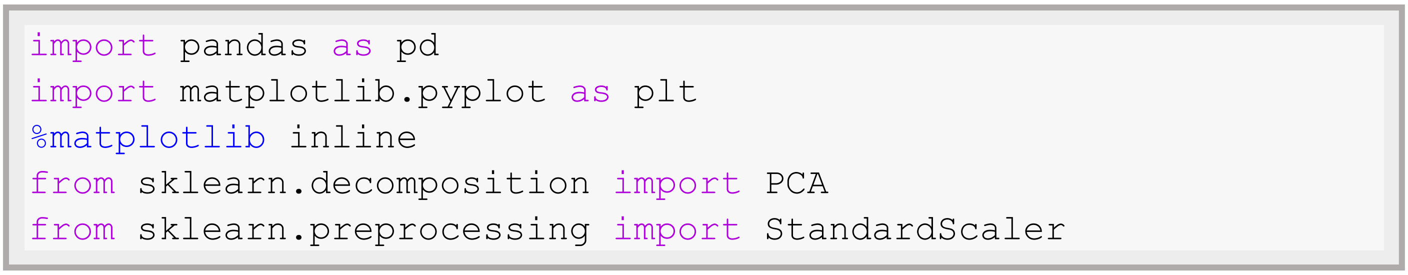
- **Dimensionality Reduction:** PCA is primarily used to reduce the number of features in a dataset while preserving the most important information. By discarding less significant principal components, it helps overcome the curse of dimensionality and improves computational efficiency.
- **Data Visualization:** PCA can be used to visualize high-dimensional data in a lower-dimensional space. By selecting the first two or three principal components, we can plot the data and gain insights into patterns, clusters, or outliers.
- **Noise Filtering:** PCA can separate the signal and noise in data. The first few principal components often capture the main signal, while the later components capture noise or less significant variations. Removing the components with lower importance can help denoise the data.
- **Feature Extraction:** PCA can be used to extract a reduced set of features from a larger feature space. These new features, represented by the principal components, can then be used as inputs to other machine learning algorithms.

Implementing PCA using Python

Dataset Required

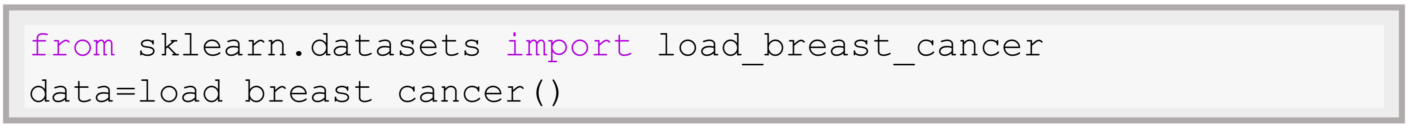
Breast Cancer dataset (loading) from sklearn datasets

Importing Required Libraries

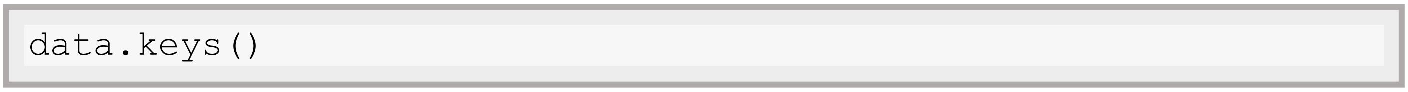


```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

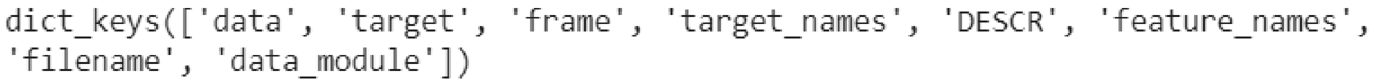
Importing (Reading) Datasets



```
from sklearn.datasets import load_breast_cancer
data=load breast cancer()
```

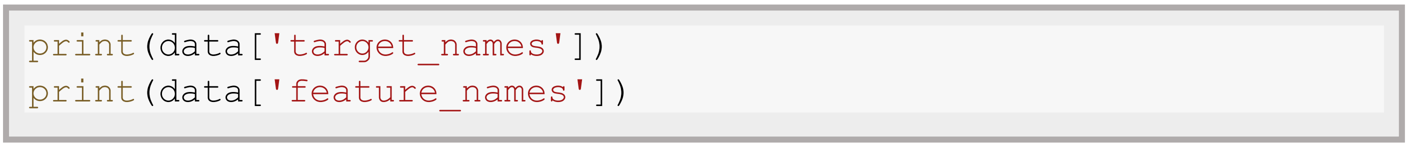


```
data.keys()
```

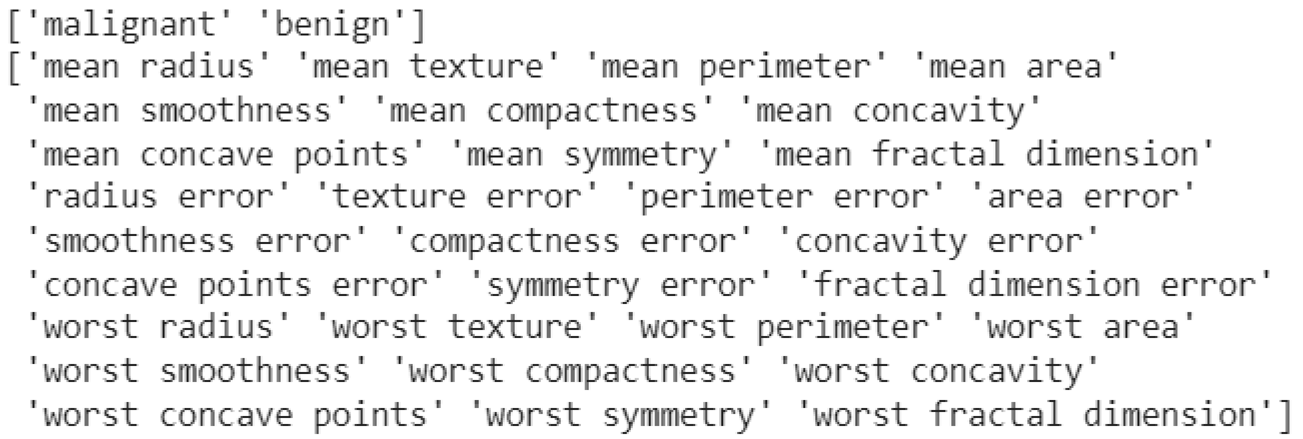


```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename', 'data_module'])
```

AssignED variables

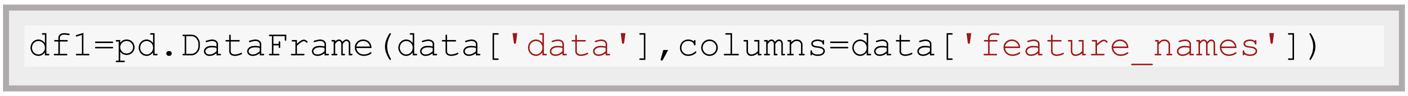


```
print(data['target_names'])
print(data['feature_names'])
```



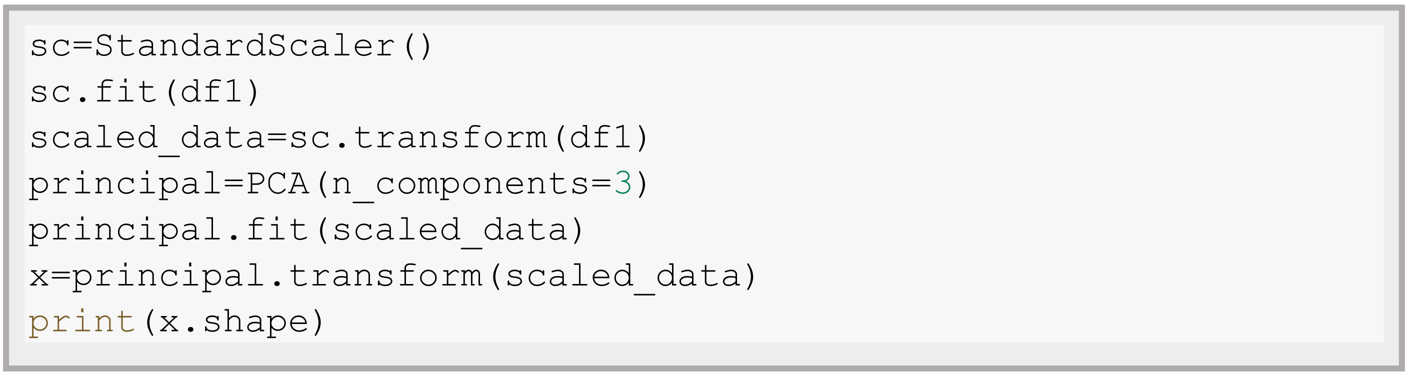
```
['malignant' 'benign']
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Creating dataframe



```
df1=pd.DataFrame(data['data'],columns=data['feature_names'])
```

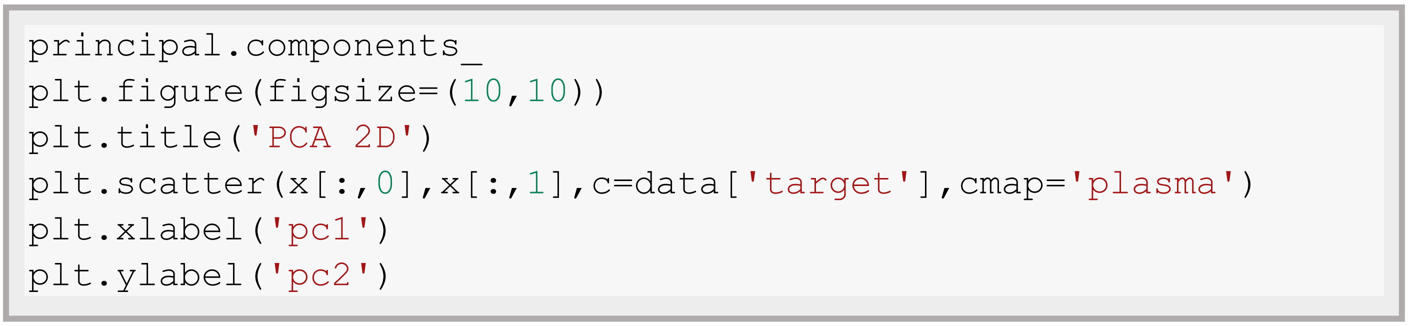
Standard Scaler and PCA



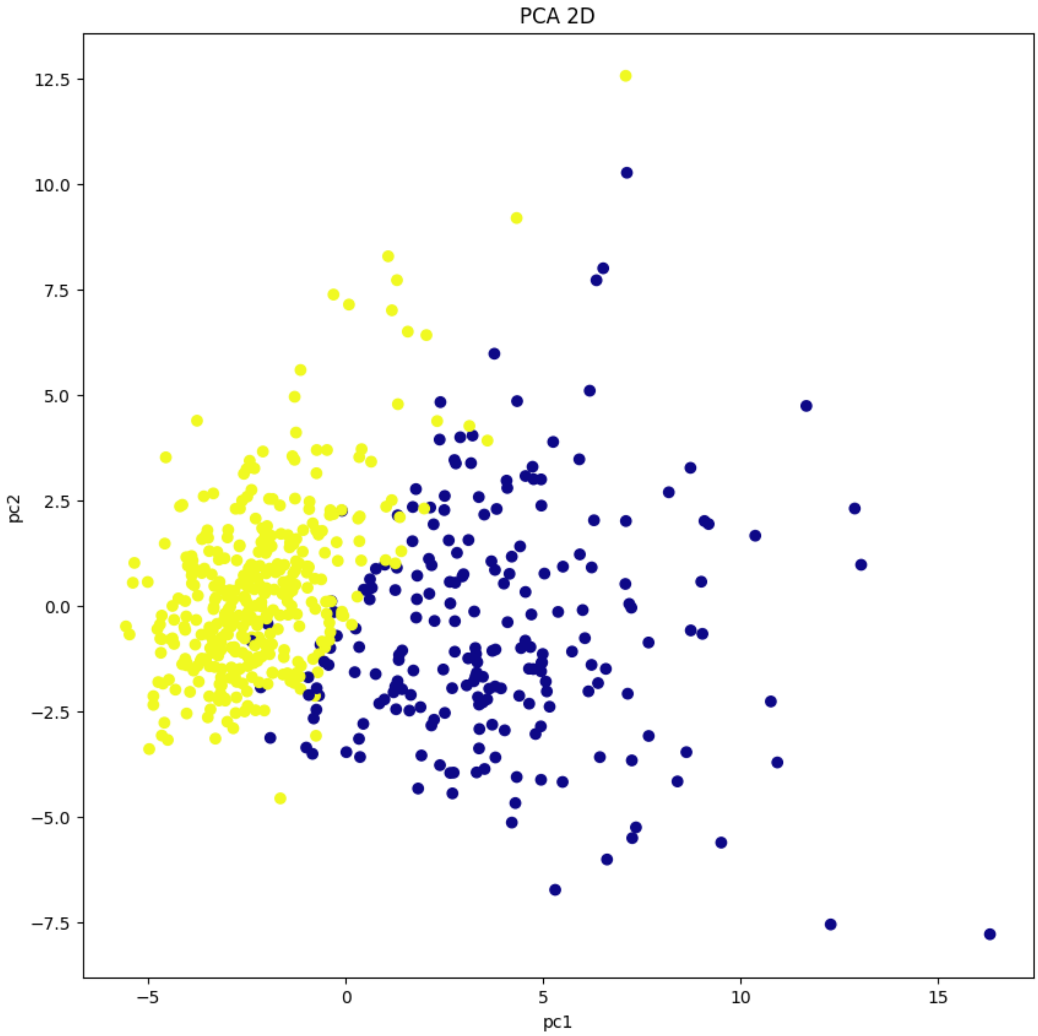
```
sc=StandardScaler()
sc.fit(df1)
scaled_data=sc.transform(df1)
principal=PCA(n_components=3)
principal.fit(scaled_data)
x=principal.transform(scaled_data)
print(x.shape)
```

```
(569, 3)
```

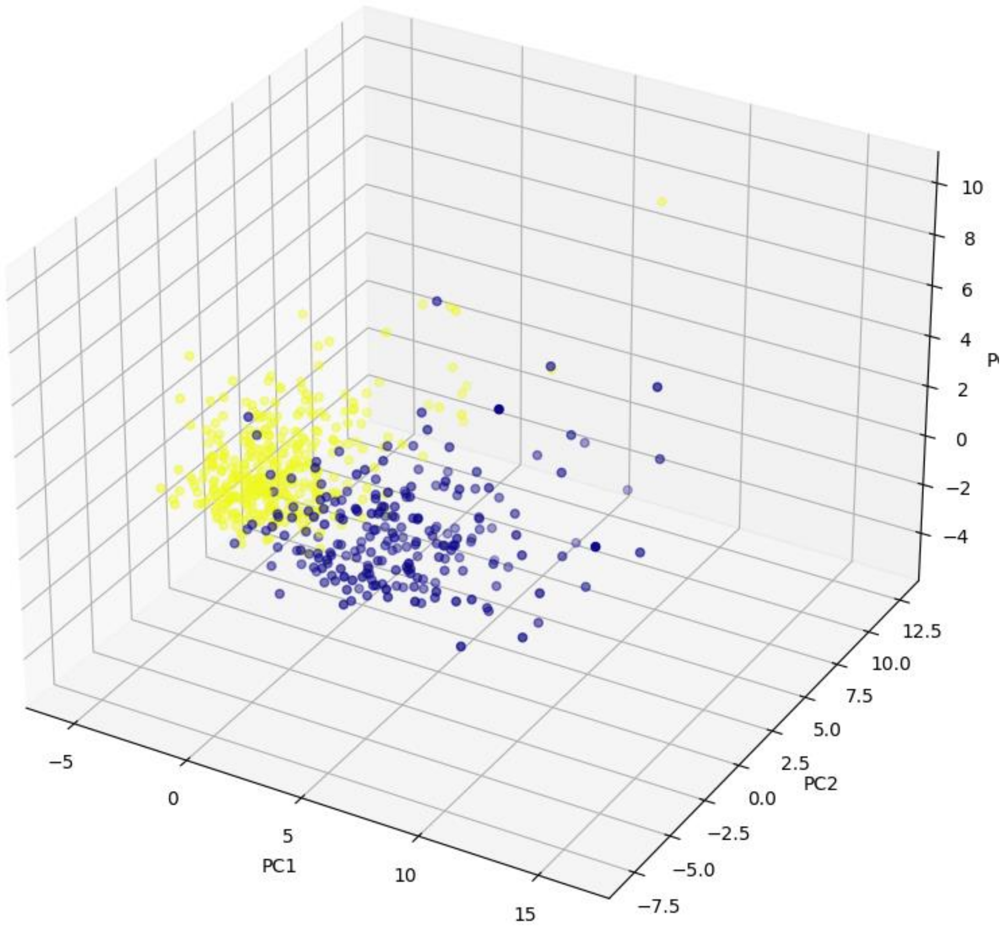
Scattering the Clusters



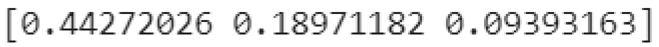
```
principal.components_
plt.figure(figsize=(10,10))
plt.title('PCA 2D')
plt.scatter(x[:,0],x[:,1],c=data['target'],cmap='plasma')
plt.xlabel('pc1')
plt.ylabel('pc2')
```



```
from mpl_toolkits.mplot3d import Axes3D
fig=plt.figure(figsize=(10,10))
axis=fig.add_subplot(111,projection='3d')
axis.scatter(x[:,0],x[:,1],x[:,2],c=data['target'],cmap='plasm
a')
axis.set_xlabel('PC1',fontsize=10)
```



```
print(principal.explained_variance_ratio_)
```

- 

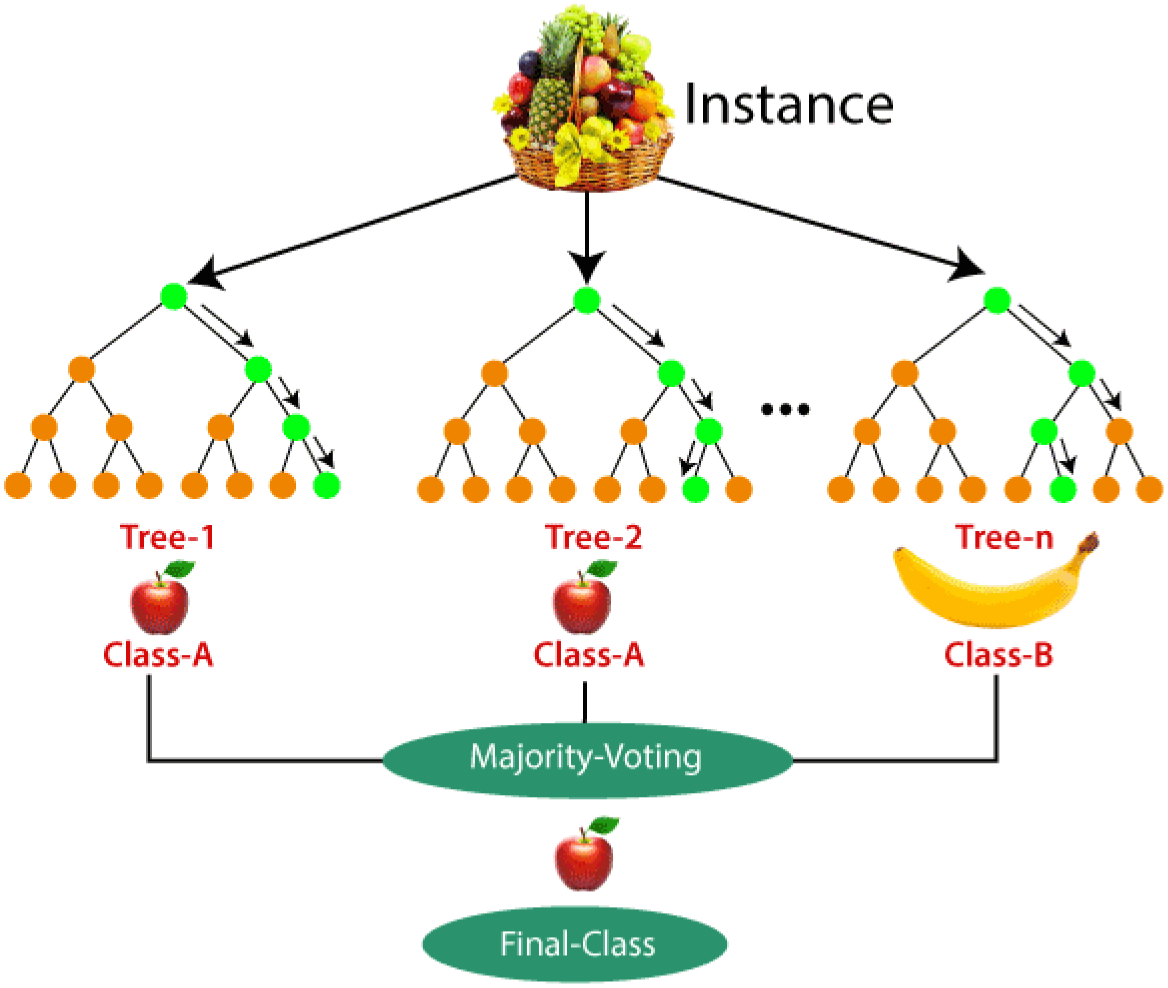
```
[0.44272026 0.18971182 0.09393163]
```

8. Random Forest

Random Forest is a popular machine learning algorithm that belongs to the ensemble learning family. It is a combination of multiple decision trees, where each tree contributes to the final prediction through a voting or averaging mechanism. Random Forest is primarily used for classification tasks, but it can also be applied to regression problems.

Random Forest builds an ensemble of decision trees by randomly selecting subsets of the training data and features. The random selection of data is called bootstrap sampling, which means that each tree is trained on a different subset of the original data created by sampling with replacement. This introduces diversity and reduces the risk of overfitting.

At each node of a decision tree, a random subset of features is considered to determine the best split. This randomness ensures that each tree has its own unique structure and avoids favoring any specific features. By combining the predictions of all the individual trees, Random Forest achieves robust and accurate results.



During the prediction phase, each tree in the Random Forest independently classifies the input data point. In the case of classification, the class that receives the majority of votes from the trees is selected as the final prediction. For regression, the average of the predictions from all the trees is taken.

Advantages of Random Forest:

- **Robustness:** Random Forest is less prone to overfitting compared to individual decision trees. The combination of multiple trees reduces the impact of noisy or irrelevant features, leading to improved generalization performance.
- **Feature Importance:** Random Forest provides a measure of feature importance, indicating the relative contribution of each feature in the classification task. This information is derived from the collective behavior of all the trees, allowing for better understanding and interpretation of the data.
- **Handling of Missing Data:** Random Forest can effectively handle missing data by using surrogate splits. It can utilize available features to make accurate predictions even when certain features have missing values.
- **Non-linearity:** Random Forest can capture complex non-linear relationships in the data. By combining multiple decision trees, it can model intricate decision boundaries and interactions between features.

Limitations of Random Forest:

- **Interpretability:** Although Random Forest can provide insights into feature importance, the final model itself is not easily interpretable. It is challenging to understand the exact logic and reasoning behind individual predictions due to the ensemble nature of the algorithm.
- **Computationally Expensive:** Random Forest can be computationally expensive, especially when dealing with a large number of trees or high-dimensional data. Training and evaluating a large ensemble of trees may require more time and computational resources.

- **Bias in Imbalanced Datasets:** Random Forest can exhibit bias towards the majority class in imbalanced datasets. Since each tree is trained independently, the majority class tends to have a stronger influence on the final predictions. Balancing techniques or specialized modifications may be required to handle imbalanced data effectively.

Applications of Random Forest:

- **Credit Scoring:** Random Forest can be used for credit scoring to assess the creditworthiness of individuals or businesses based on various financial and demographic features.
- **Disease Diagnosis:** Random Forest can assist in medical diagnosis by combining multiple factors such as patient symptoms, test results, and medical history to predict the presence or absence of a specific disease.
- **Image Recognition:** Random Forest is used in image recognition tasks, including object detection, facial recognition, and scene classification. The ensemble of decision trees can effectively analyze image features and classify them into predefined categories.
- **Fraud Detection:** Random Forest can identify fraudulent activities by analyzing patterns and anomalies in financial transactions, online behaviors, or insurance claims.
- **Customer Churn Prediction:** Random Forest can predict customer churn by considering various customer attributes, purchase history, and engagement metrics to identify customers at risk of leaving a service or product.

Implementing Random Forest using Python

Dataset Required

https://drive.google.com/file/d/1865t5MZPQn53A5bSYMN86D0BmLSBxan-/view?usp=share_link

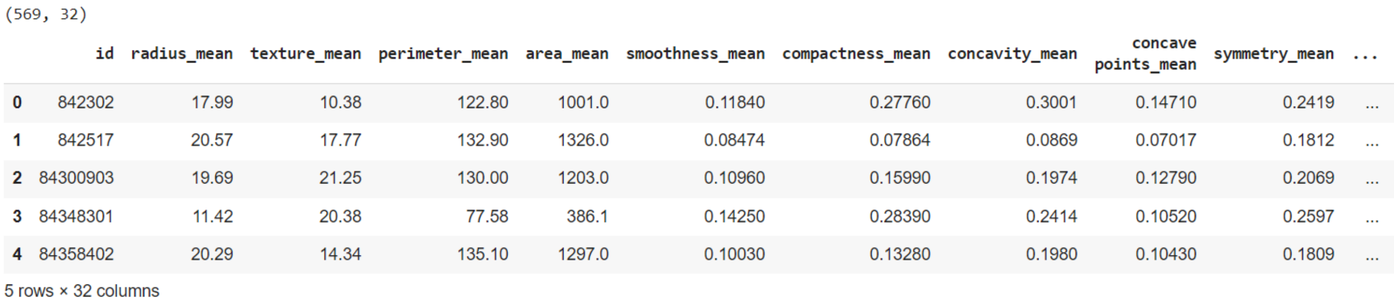
Importing Required Libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.metrics import classification_report
import seaborn as sn
import matplotlib.pyplot as plt
from sklearn import tree
```

Importing (Reading) Datasets

```
data=pd.read_csv('/content/Breast Cancer Detection
Classification Master.csv')
```

```
print(data.shape)
data.head()
```

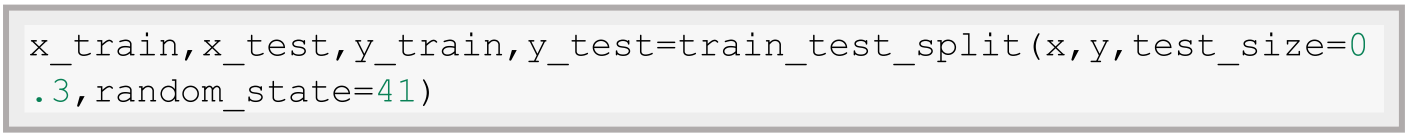


```
(569, 32)
                                                                                                                  concave
         id  radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  compactness_mean  concavity_mean  points_mean  symmetry_mean  ...
0    842302        17.99         10.38          122.80     1001.0          0.11840           0.27760          0.3001      0.14710         0.2419  ...
1    842517        20.57         17.77          132.90     1326.0          0.08474           0.07864          0.0869      0.07017         0.1812  ...
2  84300903        19.69         21.25          130.00     1203.0          0.10960           0.15990          0.1974      0.12790         0.2069  ...
3  84348301        11.42         20.38           77.58      386.1          0.14250           0.28390          0.2414      0.10520         0.2597  ...
4  84358402        20.29         14.34          135.10     1297.0          0.10030           0.13280          0.1980      0.10430         0.1809  ...
5 rows × 32 columns
```

Assigning dependent and independent variables

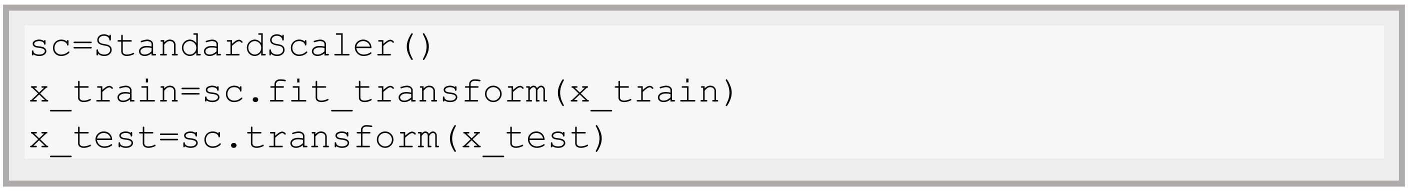
```
x=data.iloc[:,:-1].values
y=data.iloc[:,-1].values
```

Splitting the dataset into Training and Testing Dataset



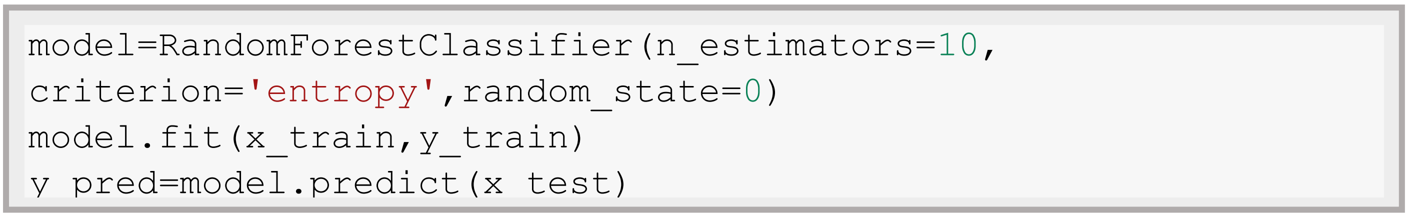
```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0
.3,random_state=41)
```

Preprocessing Data with StandardScaler



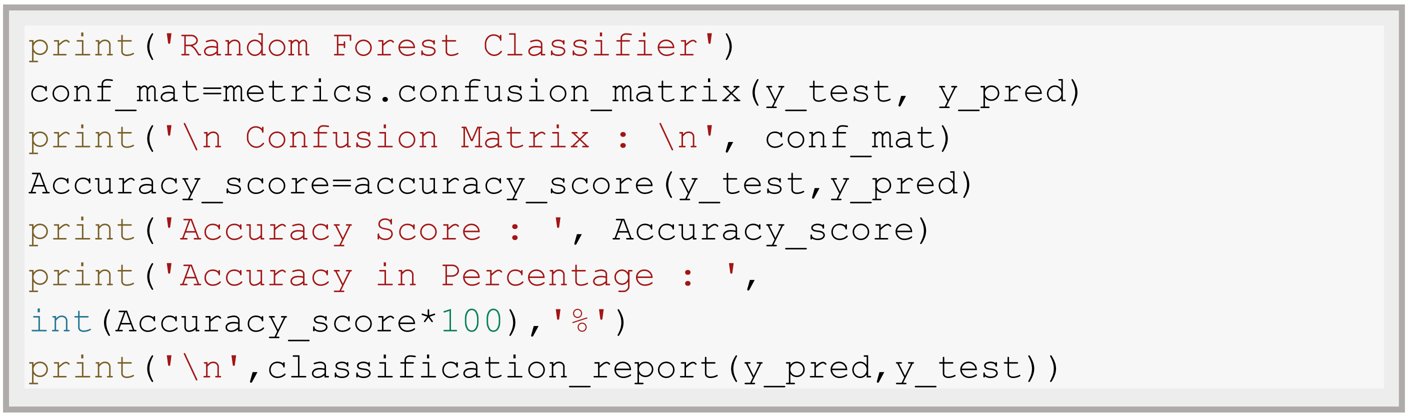
```
sc=StandardScaler()
x_train=sc.fit_transform(x_train)
x_test=sc.transform(x_test)
```

Fitting the Model Random Forest Classifier:

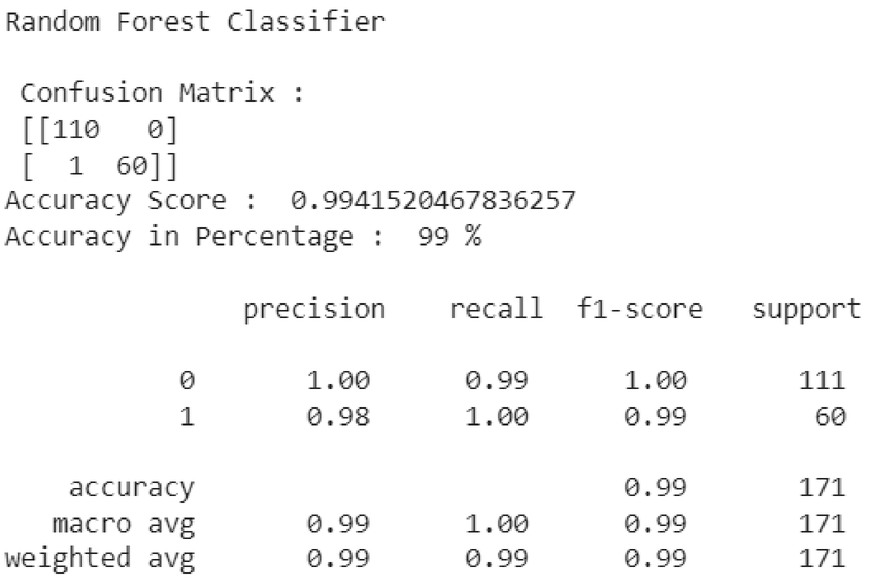


```
model=RandomForestClassifier(n_estimators=10,
criterion='entropy',random_state=0)
model.fit(x_train,y_train)
y_pred=model.predict(x_test)
```

Evaluation Metrics

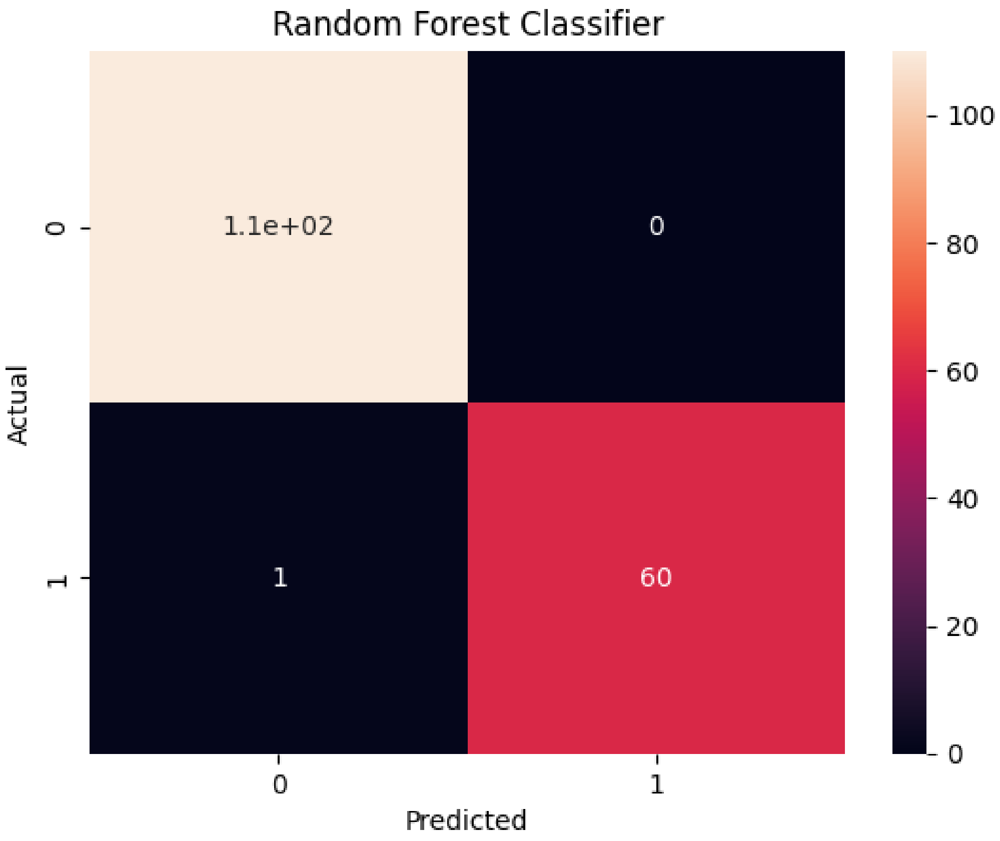


```
print('Random Forest Classifier')
conf_mat=metrics.confusion_matrix(y_test, y_pred)
print('\n Confusion Matrix : \n', conf_mat)
Accuracy_score=accuracy_score(y_test,y_pred)
print('Accuracy Score : ', Accuracy_score)
print('Accuracy in Percentage : ',
int(Accuracy_score*100),'%')
print('\n',classification_report(y_pred,y_test))
```



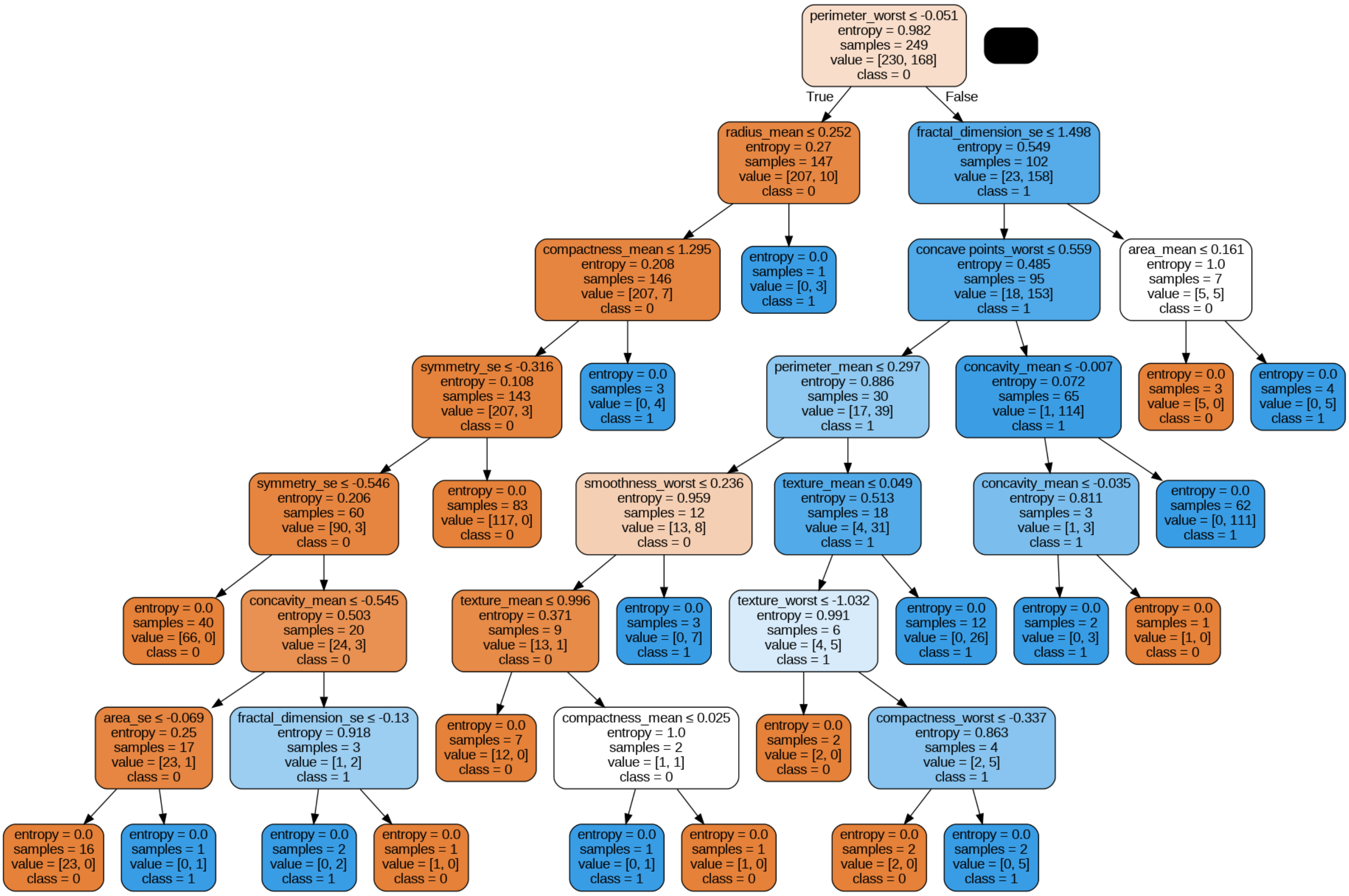
```
Random Forest Classifier

 Confusion Matrix :
 [[110   0]
 [  1  60]]
Accuracy Score :  0.9941520467836257
Accuracy in Percentage :   99 %

               precision    recall  f1-score   support

           0       1.00      0.99      1.00       111
           1       0.98      1.00      0.99        60

    accuracy                           0.99       171
   macro avg       0.99      1.00      0.99       171
weighted avg       0.99      0.99      0.99       171
```

```
conf_mat=pd.crosstab(y_test,y_pred, rownames=['Actual'],
colnames=['Predicted'])
sn.heatmap(conf_mat, annot=True).set(title='Random Forest
Classifier')
```



Visualizing Tree formed

```
import pydotplus
from IPython.display import Image

# Create DOT data
dot_data = tree.export_graphviz(model.estimators_[0],
out_file=None,
                                feature_names=data.columns[:-
1],
                                class_names=['0', '1'],
                                filled=True, rounded=True,
                                special_characters=True)

# Create graph from DOT data
graph = pydotplus.graph_from_dot_data(dot_data)

# Generate image
Image(graph.create_png())
```


perimeter_worst ≤ -0.051
entropy = 0.982
samples = 249
value = [230, 168]
class = 0
True
False
radius_mean ≤ 0.252
entropy = 0.27
samples = 147
value = [207, 10]
class = 0
fractal_dimension_se ≤ 1.498
entropy = 0.549
samples = 102
value = [23, 158]
class = 1
compactness_mean ≤ 1.295
entropy = 0.208
samples = 146
value = [207, 7]
class = 0
entropy = 0.0
samples = 1
value = [0, 3]
class = 1
concave points_worst ≤ 0.559
entropy = 0.485
samples = 95
value = [18, 153]
class = 1
area_mean ≤ 0.161
entropy = 1.0
samples = 7
value = [5, 5]
class = 0
symmetry_se ≤ -0.316
entropy = 0.108
samples = 143
value = [207, 3]
class = 0
entropy = 0.0
samples = 3
value = [0, 4]
class = 1
perimeter_mean ≤ 0.297
entropy = 0.886
samples = 30
value = [17, 39]
class = 1
concavity_mean ≤ -0.007
entropy = 0.072
samples = 65
value = [1, 114]
class = 1
entropy = 0.0
samples = 3
value = [5, 0]
class = 0
entropy = 0.0
samples = 4
value = [0, 5]
class = 1
symmetry_se ≤ -0.546
entropy = 0.206
samples = 60
value = [90, 3]
class = 0
entropy = 0.0
samples = 83
value = [117, 0]
class = 0
smoothness_worst ≤ 0.236
entropy = 0.959
samples = 12
value = [13, 8]
class = 0
texture_mean ≤ 0.049
entropy = 0.513
samples = 18
value = [4, 31]
class = 1
concavity_mean ≤ -0.035
entropy = 0.811
samples = 3
value = [1, 3]
class = 1
entropy = 0.0
samples = 62
value = [0, 111]
class = 1
entropy = 0.0
samples = 40
value = [66, 0]
class = 0
concavity_mean ≤ -0.545
entropy = 0.503
samples = 20
value = [24, 3]
class = 0
texture_mean ≤ 0.996
entropy = 0.371
samples = 9
value = [13, 1]
class = 0
entropy = 0.0
samples = 3
value = [0, 7]
class = 1
texture_worst ≤ -1.032
entropy = 0.991
samples = 6
value = [4, 5]
class = 1
entropy = 0.0
samples = 12
value = [0, 26]
class = 1
entropy = 0.0
samples = 2
value = [0, 3]
class = 1
entropy = 0.0
samples = 1
value = [1, 0]
class = 0
area_se ≤ -0.069
entropy = 0.25
samples = 17
value = [23, 1]
class = 0
fractal_dimension_se ≤ -0.13
entropy = 0.918
samples = 3
value = [1, 2]
class = 1
entropy = 0.0
samples = 7
value = [12, 0]
class = 0
compactness_mean ≤ 0.025
entropy = 1.0
samples = 2
value = [1, 1]
class = 0
entropy = 0.0
samples = 2
value = [2, 0]
class = 0
compactness_worst ≤ -0.337
entropy = 0.863
samples = 4
value = [2, 5]
class = 1
entropy = 0.0
samples = 16
value = [23, 0]
class = 0
entropy = 0.0
samples = 1
value = [0, 1]
class = 1
entropy = 0.0
samples = 2
value = [0, 2]
class = 1
entropy = 0.0
samples = 1
value = [1, 0]
class = 0
entropy = 0.0
samples = 1
value = [0, 1]
class = 1
entropy = 0.0
samples = 1
value = [1, 0]
class = 0
entropy = 0.0
samples = 2
value = [2, 0]
class = 0
entropy = 0.0
samples = 2
value = [0, 5]
class = 1

9. Time Series Modeling

Time series modeling is a statistical technique used to analyze and forecast data that varies over time. It is particularly suited for datasets where the order and timing of observations are important. Time series data is commonly encountered in fields such as finance, economics, weather forecasting, and stock market analysis.

The fundamental concept in time series modeling is that the observations at different time points are not independent, but rather exhibit temporal dependence. The goal is to capture and model the underlying patterns, trends, and seasonal variations in the data to make accurate predictions or forecasts.

Main Components of Time series:

1. **Trend:** It represents the long-term direction or pattern in the data. It can be increasing (upward trend), decreasing (downward trend), or remain relatively constant (horizontal trend).
2. **Seasonality:** It refers to regular and recurring patterns that occur at fixed intervals, such as daily, weekly, or yearly cycles. Seasonality can be influenced by various factors like time of year, holidays, or weather conditions.
3. **Residuals (or noise):** It represents the random fluctuations or irregularities that cannot be explained by the trend or seasonality. Residuals are typically assumed to follow a random or stochastic process.

The most common approach to time series modeling is using autoregressive integrated moving average (ARIMA) models.

ARIMA models combine three components:

1. **Autoregressive (AR) component:** It models the relationship between an observation and a linear combination of its past values. It captures the effect of previous observations on the current value.

2. **Integrated (I) component:** It deals with the process of differencing the time series data to achieve stationarity. Stationarity is a desirable property in time series modeling, where the mean, variance, and autocorrelation structure remain constant over time.
3. **Moving average (MA) component:** It models the relationship between an observation and a linear combination of past error terms. It captures the residual fluctuations that are not accounted for by the autoregressive component.

Advantages of Time Series Modeling:

- **Forecasting:** Time series modeling enables accurate forecasting of future values based on historical patterns and trends. It provides valuable insights for planning, decision-making, and resource allocation.
- **Trend and Seasonality Analysis:** Time series modeling helps in identifying and understanding long-term trends and seasonal patterns in the data. This information can be used to detect and explain changes in the underlying process.
- **Impact Assessment:** Time series models can be used to assess the impact of specific events or interventions on the data. It helps in evaluating the effectiveness of policies, marketing campaigns, or other interventions.

Limitations of Time Series Modeling:

- **Stationarity Assumption:** Most time series models assume that the underlying process is stationary, which may not always hold true in real-world scenarios. Non-stationarity can introduce biases and affect the accuracy of forecasts.
- **Limited Extrapolation:** Time series models are generally good at forecasting short-term patterns but may struggle with long-term extrapolation. The accuracy of predictions tends to decrease as the forecast horizon increases.
- **Sensitivity to Outliers:** Time series models can be sensitive to outliers or unusual observations, which can disproportionately influence the model's estimates and predictions. Outliers need to be handled carefully to avoid biased results.

Applications of Time Series Modeling:

- **Economic Forecasting:** Time series models are extensively used in economics to forecast economic indicators like GDP, inflation rates, stock prices, and interest rates. They help policymakers, investors, and analysts make informed decisions.
- **Demand Forecasting:** Time series modeling is valuable in predicting product demand, allowing companies to optimize inventory management, production planning, and supply chain operations.
- **Energy Load Forecasting:** Time series modeling is used to forecast energy demand and load patterns, helping energy providers optimize energy generation, distribution, and pricing strategies. Accurate load forecasting is crucial for maintaining grid stability and meeting customer demand.
- **Weather Forecasting:** Time series models are employed in weather forecasting to predict variables such as temperature, precipitation, wind speed, and humidity. These forecasts are vital for various sectors, including agriculture, transportation, and disaster management.
- **Stock Market Analysis:** Time series modeling assists in analyzing stock price movements and identifying trends and patterns. It helps investors and traders make informed decisions regarding buying, selling, and portfolio management.
- **Sales Forecasting:** Time series modeling is utilized in sales forecasting to predict future sales volumes or revenues. This information aids businesses in production planning, inventory management, and resource allocation.
- **Epidemiology and Disease Surveillance:** Time series models are valuable for tracking and predicting disease outbreaks, analyzing epidemic patterns, and estimating the spread of infectious diseases. They play a crucial role in public health decision-making and resource allocation.
- **Financial Market Analysis:** Time series modeling is employed in analyzing financial market data, such as exchange rates, interest rates, and commodity prices. It assists in identifying trends, seasonality, and volatility patterns, aiding in investment strategies and risk management.

- **Quality Control:** Time series modeling is used to monitor and control manufacturing processes, ensuring product quality and identifying deviations from standard performance. It helps in maintaining consistency and minimizing defects.
- **Web Traffic Analysis:** Time series modeling helps analyze web traffic patterns, predict website visitor volumes, and optimize server resources and capacity planning.

Implementing Random Forest using Python

Dataset Required

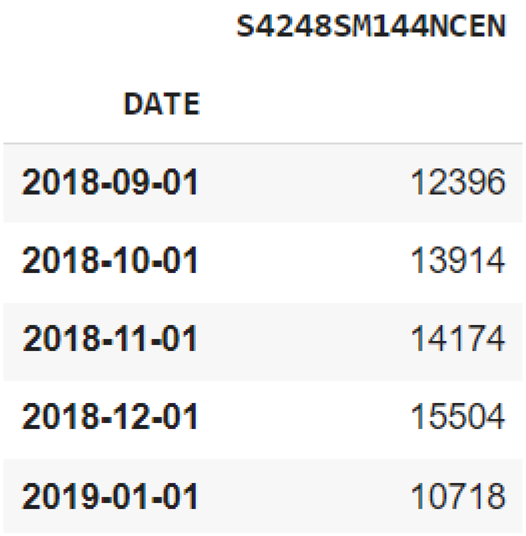
https://drive.google.com/file/d/1s-aZWEqNCPTBWCK6qACmbqH6lMDt6Hyp/view?usp=sharing

Importing Required Libraries

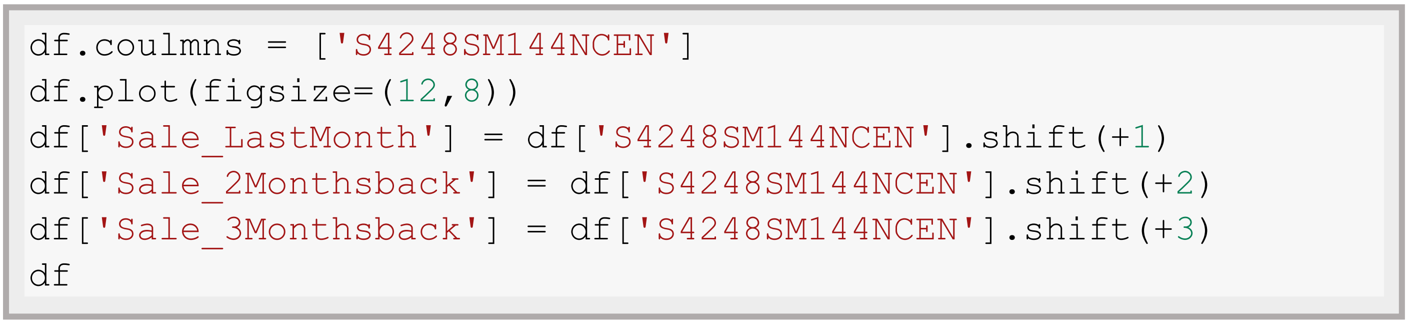
```python
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from math import sqrt
```

Importing (Reading) Datasets

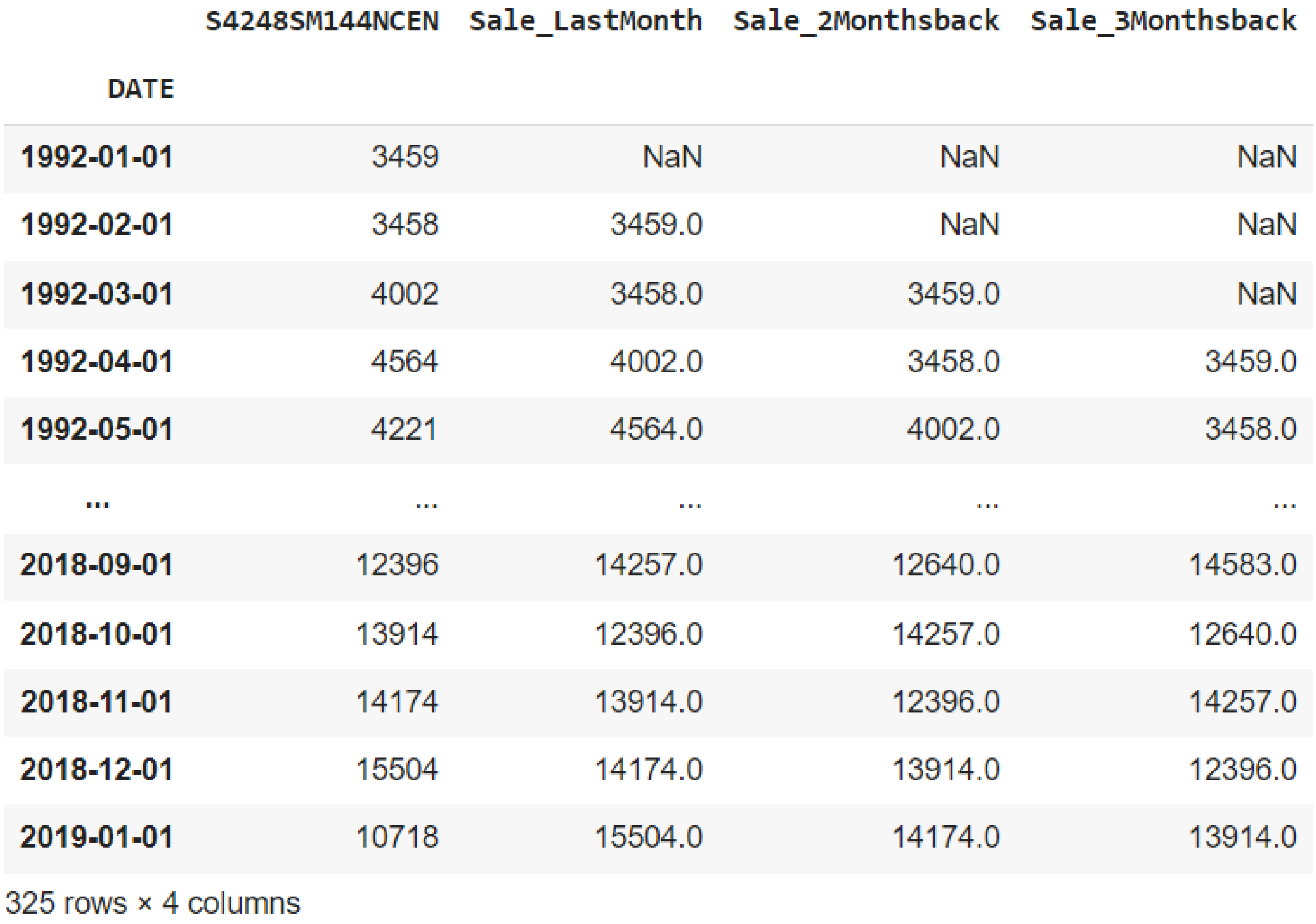
```python
df = pd.DataFrame()
df = pd.read_csv('Alcohol_Sales.csv',index_col =
'DATE',parse_dates = True)
df.index.freq = 'MS'
df.tail()
```



	S4248SM144NCEN
DATE	
2018-09-01	12396
2018-10-01	13914
2018-11-01	14174
2018-12-01	15504
2019-01-01	10718

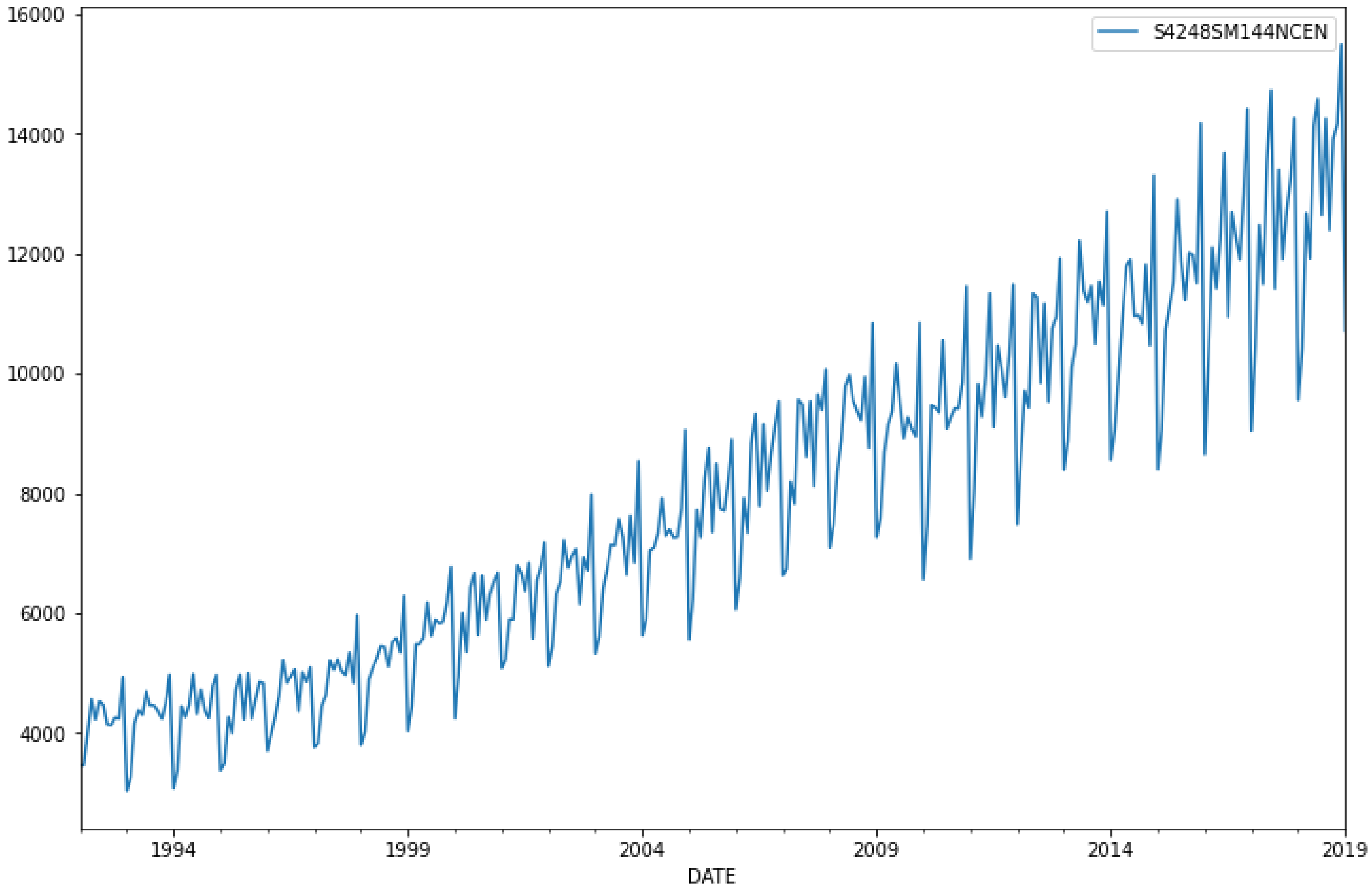


```
df.coulmns = ['S4248SM144NCEN']
df.plot(figsize=(12,8))
df['Sale_LastMonth'] = df['S4248SM144NCEN'].shift(+1)
df['Sale_2Monthsback'] = df['S4248SM144NCEN'].shift(+2)
df['Sale_3Monthsback'] = df['S4248SM144NCEN'].shift(+3)
df
```

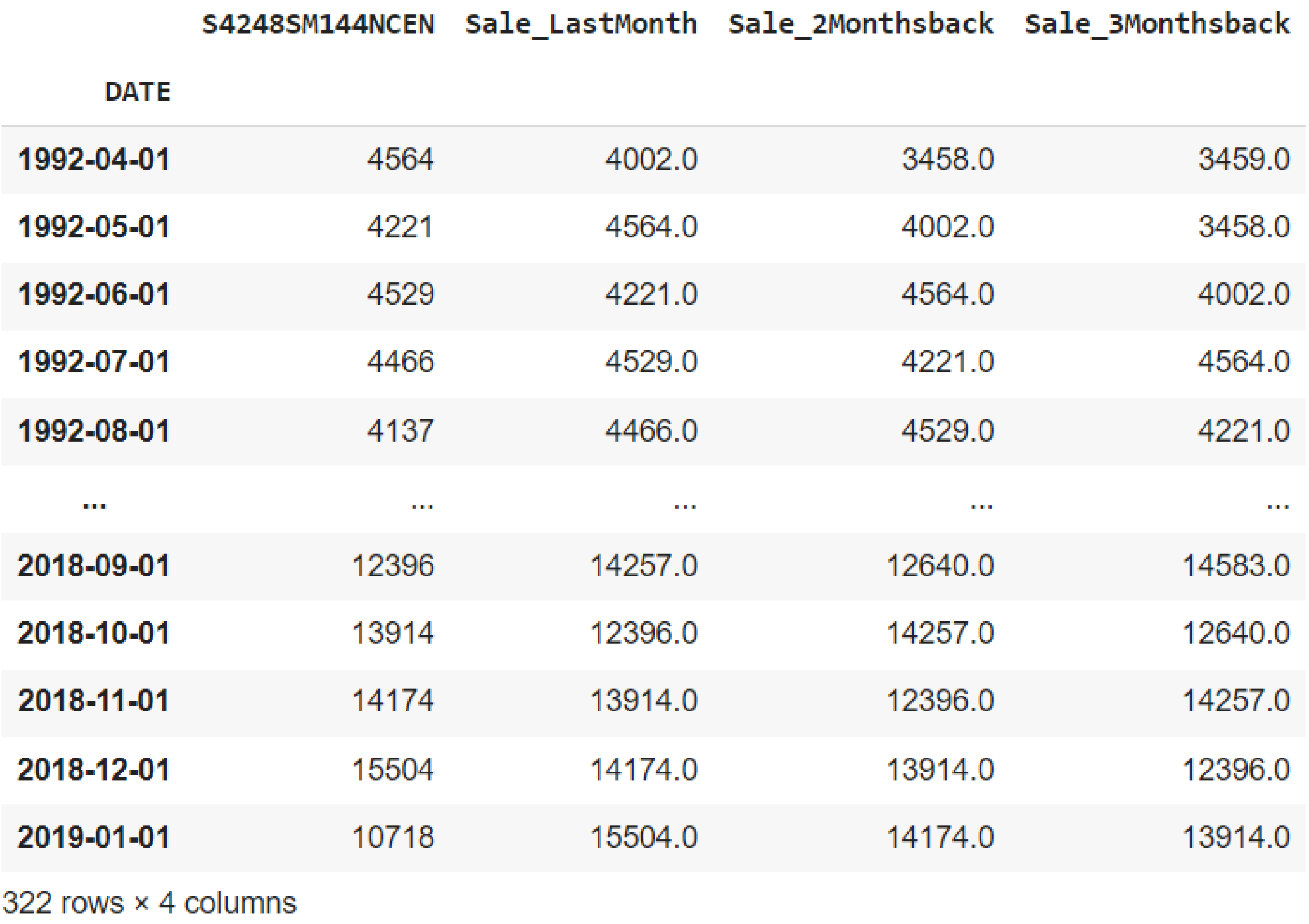


	S4248SM144NCEN	Sale_LastMonth	Sale_2Monthsback	Sale_3Monthsback
DATE				
1992-01-01	3459	NaN	NaN	NaN
1992-02-01	3458	3459.0	NaN	NaN
1992-03-01	4002	3458.0	3459.0	NaN
1992-04-01	4564	4002.0	3458.0	3459.0
1992-05-01	4221	4564.0	4002.0	3458.0
...	...	...	...	...
2018-09-01	12396	14257.0	12640.0	14583.0
2018-10-01	13914	12396.0	14257.0	12640.0
2018-11-01	14174	13914.0	12396.0	14257.0
2018-12-01	15504	14174.0	13914.0	12396.0
2019-01-01	10718	15504.0	14174.0	13914.0

325 rows × 4 columns


S4248SM144NCEN
16000
14000
12000
10000
8000
6000
4000
1994
1999
2004
2009
2014
2019
DATE

```
df = df.dropna()
df
```



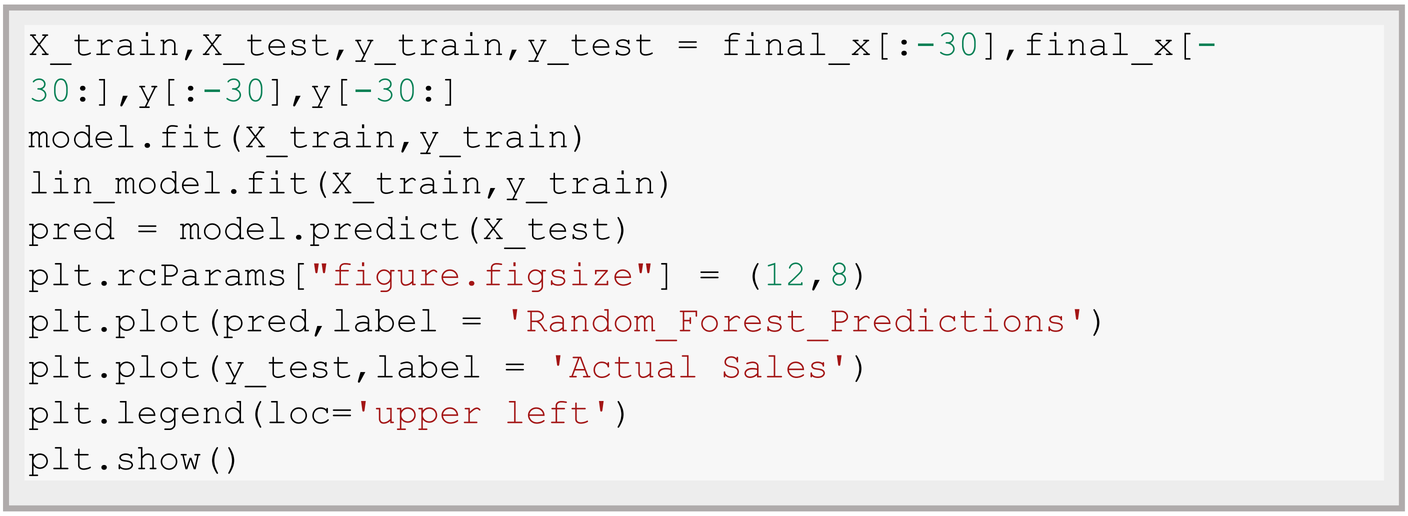
	S4248SM144NCEN	Sale_LastMonth	Sale_2Monthsback	Sale_3Monthsback
DATE				
1992-04-01	4564	4002.0	3458.0	3459.0
1992-05-01	4221	4564.0	4002.0	3458.0
1992-06-01	4529	4221.0	4564.0	4002.0
1992-07-01	4466	4529.0	4221.0	4564.0
1992-08-01	4137	4466.0	4529.0	4221.0
...	...	...	...	...
2018-09-01	12396	14257.0	12640.0	14583.0
2018-10-01	13914	12396.0	14257.0	12640.0
2018-11-01	14174	13914.0	12396.0	14257.0
2018-12-01	15504	14174.0	13914.0	12396.0
2019-01-01	10718	15504.0	14174.0	13914.0

322 rows × 4 columns

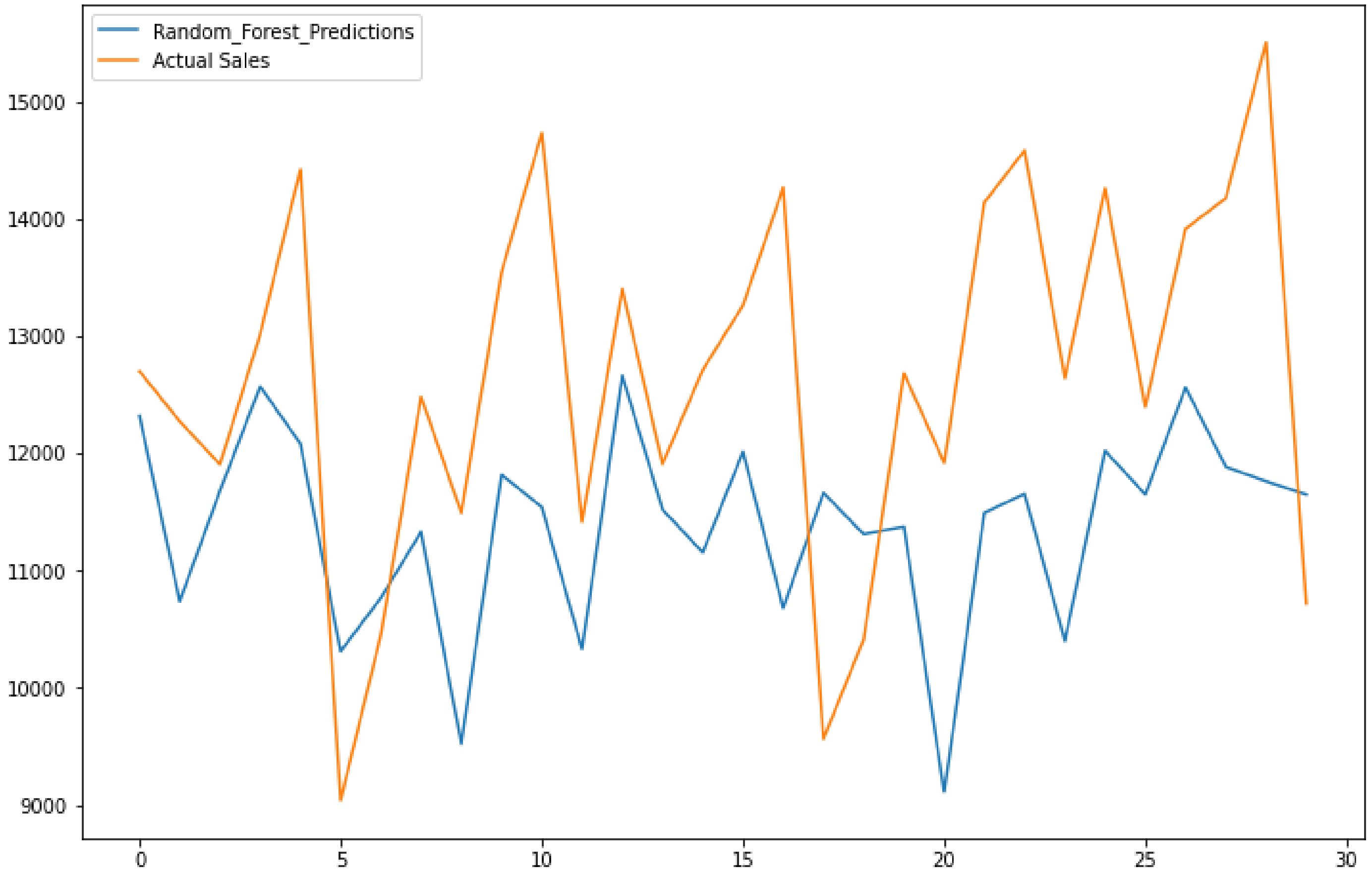
Fitting using Linear Regression and Random Forest Classifier

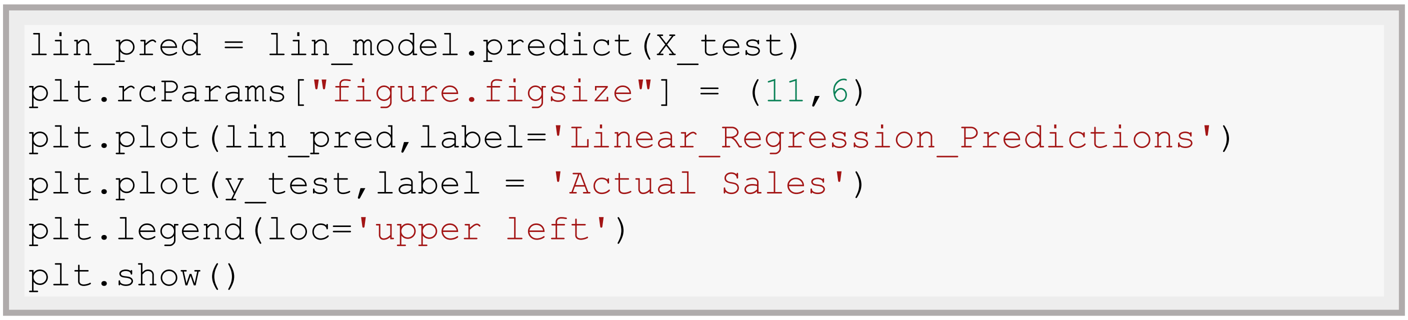
```
lin_model = LinearRegression()
model = RandomForestRegressor(n_estimators = 100,max_features
= 3,random_state = 1)
```

```
x1,x2,x3,y =
df['Sale_LastMonth'],df['Sale_2Monthsback'],df['Sale_3Monthsba
ck'],df['S4248SM144NCEN']
x1,x2,x3,y =
np.array(x1),np.array(x2),np.array(x3),np.array(y)
x1,x2,x3,y = x1.reshape(-1,1),x2.reshape(-1,1),x3.reshape(-
1,1),y.reshape(-1,1)
final_x = np.concatenate((x1,x2,x3),axis = 1)
print(final_x)
```

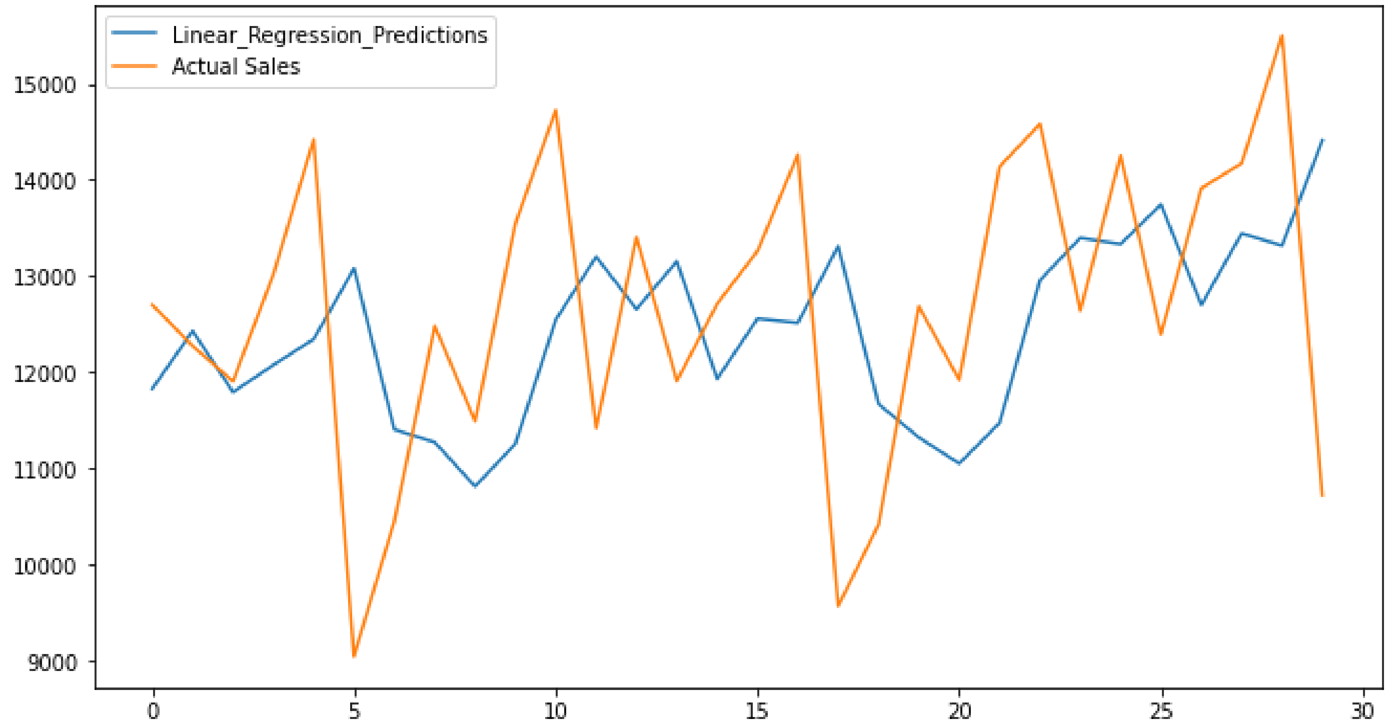


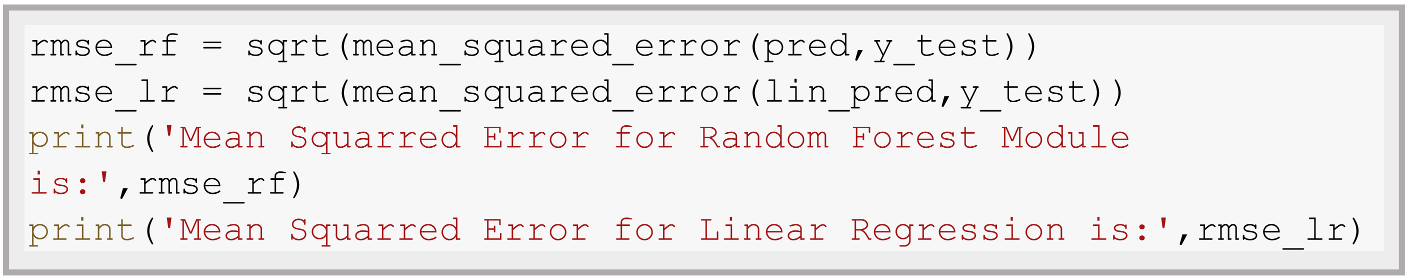
```
X_train,X_test,y_train,y_test = final_x[:-30],final_x[-
30:],y[:-30],y[-30:]
model.fit(X_train,y_train)
lin_model.fit(X_train,y_train)
pred = model.predict(X_test)
plt.rcParams["figure.figsize"] = (12,8)
plt.plot(pred,label = 'Random_Forest_Predictions')
plt.plot(y_test,label = 'Actual Sales')
plt.legend(loc='upper left')
plt.show()
```





```
lin_pred = lin_model.predict(X_test)
plt.rcParams["figure.figsize"] = (11,6)
plt.plot(lin_pred,label='Linear_Regression_Predictions')
plt.plot(y_test,label = 'Actual Sales')
plt.legend(loc='upper left')
plt.show()
```





```
rmse_rf = sqrt(mean_squared_error(pred,y_test))
rmse_lr = sqrt(mean_squared_error(lin_pred,y_test))
print('Mean Squarred Error for Random Forest Module
is:',rmse_rf)
print('Mean Squarred Error for Linear Regression is:',rmse_lr)
```

```
Mean Squarred Error for Random Forest Module is: 1913.7762399350665
Mean Squarred Error for Linear Regression is: 1791.496523275983
```